НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.9

«До захисту допущено»
Завідувач кафедри СПСКС

_____Віталій РОМАНКЕВИЧ
(підпис) (ім'я, прізвище)
"___"_____2020р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія

на тему: Засоби підвищення ефективності сервісу електронної черги_____
_____
_____

Виконав: студент ІІ курсу, групи КВ-91мп
Харитончик Олександр Вячеславович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник к.т.н. доц. СП і СКС Клятченко Я.М.____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю доцент, с.н.с.,к.т.н. Юлія БОЯРІНОВА __
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

за освітньо-професійною програмою

Спеціальність 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
«___»_____2019 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Харитончик Олександр Вячеславович
(прізвище, ім'я, по батькові)

1. Тема дисертації «Засоби підвищення ефективності сервісу електронної черги»,
науковий керівник дисертації к.т.н. Клятченко Ярослав Михайлович,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «12» листопада 2020 р. № 3298-С

2. Термін подання студентом дисертації 10 грудня 2020 р.

3. Об'єкт дослідження є процеси обробки запитів клієнтів, що бажають стати у чергу.

4. Предмет дослідження розробка сервісу обробки запитів

5. Перелік завдань, які потрібно розробити провести аналіз існуючих методів

розв’язання задачі, вибрати метод розв'язання системи, спроектувати автоматизовану систему електронної черги, що зменшить час очікування у черзі та надасть можливість записатися у чергу віддалено, за допомогою власнорозробленого сервісу, виконати програмну реалізацію розробленої системи, виконати тестування розробленої системи.

6. Перелік ілюстративного матеріалу - презентація, скріншоти роботи сервісу

7. Перелік публікацій - «Сервіс електронної черги», наукова конференція магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2020 (Київ, 18-19 листопада 2020 рік);
«Сервіс електронної черги», VII Міжнародна науково-технічна Internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційнотехнічними та технологічними комплексами» (Національний Університет Харчових Технологій, 26 листопада 2020 рік)

8. Дата видачі завдання 5 листопада 2019 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою дисертації	10.11.2019	
2.	Підготовка матеріалів першого розділу магістерської дисертації	12.12.2019	
3.	Підготовка матеріалів другого розділу магістерської дисертації	14.02.2020	
4.	Підготовка матеріалів третього розділу магістерської дисертації	19.04.2020	
5.	Підготовка матеріалів четвертого розділу магістерської дисертації	05.05.2020	
6.	Розробка програмного забезпечення	30.06.2020	
7.	Тестування програмного забезпечення	19.07.2020	
8.	Попередній розгляд магістерської дисертації на кафедрі	27.11.2020	

Студент _____ Олександр ХАРИТОНЧИК
(підпис)

Науковий керівник дисертації _____ Ярослав КЛЯТЧЕНКО
(підпис)

РЕФЕРАТ

Актуальність теми. Наразі актуальність досить велика, оскільки тільки завдяки такому сервісу користувачі зможуть самостійно записатися у чергу будь-куди та будь-коли без зайвого клопоту. Для цього Вам знадобиться лише доступ до мережі інтернет. Ми живемо у дуже швидкоплинному світі, особливо якщо ви живете у великому місті, тому без навичок економії часу на зайві дії та його раціонального використання нам ніяк не впоратися, а даний застосунок якнайкраще економить час та гроші як клієнтів, так і власників якоїсь установи, куди необхідно стати у чергу.

Об’єктом дослідження є процеси обробки запитів клієнтів, що бажають стати у чергу.

Предметом дослідження є розробка сервісу обробки запитів.

Метою роботи є підвищення ефективності програмно-апаратних засобів сервісу електронної черги за рахунок зменшення часу очікування у черзі та реалізації нових функціональних можливостей дистанційної взаємодії із засобами цього сервісу.

Методи дослідження. Аналіз існуючих сервісів. Аналіз існуючих методів розв’язання задачі. Вибір методу розв'язання системи. Проектування автоматизованої систему електронної черги, що зменшить час очікування у черзі та надасть можливість записатися у чергу віддалено, за допомогою власно розробленого сервісу. Виконання програмної реалізацію розробленої системи. Виконання тестування розробленої системи.

Наукова новизна. Запропоновано новий спосіб реалізації сервісів електронних черг, що дозволяє покращити такі конкурентні властивості сервісу як універсальність та незалежність від структур даних .

Практична цінність полягає у можливості легкого додавання будь-якого місця на яке користувач бажає стати у чергу та використання сервісу з будь-якого пристрою, що може під’єднатися до мережі інтернет за допомогою браузера.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі представлена проблемна область, мета та актуальність роботи.

У першому розділі описується постановка задачі.

У другому розділі описується теоретичне підґрунтя реалізації.

У третьому розділі описуються особливості архітектурно-структурної організації.

У четвертому розділі наведено опис та аналіз отриманих результатів.

У висновках стисло наводяться результати розробки та досліджень.

Ключові слова сервіс, алгоритм, програмне забезпечення, черга, сервіс масового обслуговування.

РЕФЕРАТ

Актуальность темы. На текущий момент актуальность достаточно велика, поскольку только благодаря такому сервису пользователь сможет самостоятельно записаться в очередь куда угодно и когда угодно без лишних хлопот. Для этого ему понадобится лишь доступ к сети интернет. Мы живем в очень скоротечном мире, особенно если вы живете в большом городе, поэтому без навыков экономии времени на лишние действия и его рационального использование нам никак не справиться, а данное приложение помогает экономить время и деньги как клиентов, так и собственников какого-то учреждения, куда необходимо стать в очередь.

Объектом исследования являются процессы обработки запросов клиентов, которые хотят стать в очередь.

Предметом исследования является разработка сервиса обработки запросов.

Целью работы является повышение эффективности программно-аппаратных средств сервиса электронной очереди за счет уменьшения времени ожидания в очереди и реализации новых функциональных возможностей дистанционного взаимодействия со средствами этого сервиса.

Методы исследования. Анализ существующих сервисов. Анализ существующих методов решения задачи. Выбор метода решения системы. Проектирование автоматизированной системы электронной очереди, которая уменьшит время ожидания в очереди и предоставит возможность записаться в очередь удаленно, с помощью собственно разработанного сервиса. Выполнение программной реализации разработанной системы. Выполнение тестирования разработанной системы.

Научная новизна. Предложено новый способ реализации сервисов электронных очередей, что позволяет улучшить такие конкурентные свойства сервиса как универсальность и независимость от структур данных.

Практическая ценность состоит в возможности легкого добавления любого места на которое пользователь пользователь желает стать в очередь и использования сервиса с любого которого устройства, которое может подключаться к сети интернет с помощью браузера.

Структура и объем работы. Магистерская диссертация состоит из вступления, четырех разделов, выводов и приложений.

Во введении представлена проблемная область, цель и актуальность работы.

В первом разделе описывается постановка задачи.

В втором разделе описываются теоретические основы реализации.

У третьем разделе описывается особенности архитектурно-структурной организации.

В четвертом разделе приведено описание и анализ полученных результатов.

В выводах кратко показываются результаты разработки и исследований.

Ключевые слова сервис, алгоритм, программное обеспечение, очередь, сервис массового обслуживания.

ABSTRACT

Actuality of the topic. At the moment, the actuality is quite large, because only with the help of such a service, the user will be able to independently enroll in the queue anywhere and anytime without unnecessary hassle. To do this, they only needs access to the Internet. We live in a very fast-moving world, especially if you live in a big city, therefore, without the skills time-saving for unnecessary actions and its rational use, we cannot cope, and this application helps to save time and money for both clients and owners of an institution where you want to queue.

The object of the research is the processes of processing customer requests who want to queue.

The subject of the research is the development of a request processing service.

The aim of the work is to increase the efficiency of the software and hardware of the electronic queue service by reducing the waiting time in the queue and implementing new functionality for remote interaction with the means of this service.

Research methods. Analysis of existing services. Analysis of existing methods for solving the problem. The choice of the method for solving the system. Designing an automated electronic queue system that will reduce the waiting time in the queue and provide an opportunity to enroll in the queue remotely, using a properly developed service. Implementation of the developed system. Testing the developed system.

Scientific novelty. A new way of implementing electronic queuing services is proposed, which makes it possible to improve such competitive properties of the service as universality and independence from data structures.

The practical value lies in the ability to easily add any place to which the user wants to queue and use the ability to use service from any device that can connect to the Internet using a browser.

Structure and scope of work. The master's thesis consists of an introduction, four chapters, conclusions and applications.

The introduction presents the problem area, purpose and relevance of the work.

The first chapter describes the problem statement.

The second chapter describes the theoretical foundations of the implementation.

The third chapter describes the features of the architectural and structural organization.

The fourth chapter describes and analyzes the results obtained.

The findings summarize the results of development and research.

Keywords service, algorithm, software, queue, queuing service.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД — база даних.

ДМСУ — державна міграційна служба України.

ПЗ — програмне забезпечення.

ПК — персональний комп’ютер.

СМО — система масового обслуговування.

СКБД — система контролю бази даних.

API — application programming interface.

CRUD — create, read, update, delete.

Gdpr — general data protection regulation.

HTML — hypertext markup language.

HTTP — hypertext transfer protocol.

HTTPS — hypertext transfer protocol secure.

JPEG — joint photographic experts group.

JSON — javascript object notation.

REST — representational state transfer.

SOAP — simple object access protocol.

SQL — structured query language.

URL — uniform resource locator.

XML — eXtensible markup language.

XSS — cross-site scripting.

ВСТУП

На теперішній момент діє карантин вихідного дня, це означає, що у вихідні дні всі кафе, ресторани, торгово-розважальні центри та інші подібні типи місць, де можуть виникати великі скупчення людей, не працюють. Це значно збільшує ажіотаж серед клієнтів у будні дні, що навпаки призводить до ще більших утворень натовпів у післяробочий час.

Люди, що мають у планах відвідати декілька місць після своєї роботи мають усього лише декілька годин до закриття магазинів, щоб дістатися до них та отримати бажані послуги. У такому випадку вони вимушені обирати лише одне місце на день через брак часу. Нерідко трапляються випадки коли людина дістається потрібного магазину чи кафе та розуміє, що не встигне отримати те, за чим прийшла через великий обсяг охочий, що прийшли раніше, наприклад в кафе можуть впускати лише по двадцять людей, а черги у телефонному режимі не формуються. У такі випадки вона розуміє, що могла б піти у інше місце, але часу вже немає. В такі моменти дуже не вистачає якогось місця, де можна було б подивитися усі наявні черги та стати у відповідну за бажанням.

Наразі актуальність досить велика, оскільки тільки завдяки такому сервісу ви зможете самостійно записатися у чергу будь-куди та будь-коли без зайвого клопоту. Для цього Вам знадобиться лише доступ до мережі інтернет. Ми живемо у дуже швидкоплинному світі, особливо якщо ви живете у великому місті, тому без навичок економії часу на зайві дії та його раціонального використання нам ніяк не впоратися, а даний застосунок дозволяє економити час та гроші як клієнтів, так і власників якоїсь установи, куди необхідно стати у чергу [1].

Проблемною областю магістерської дисертації є дослідження підходів до розв’язку проблеми запису людей у чергу та відстеження свого місця у черзі та

дотримання порядку черги.

Враховуючи все вищесказане можемо поставити задачу, а саме: розробити нову систему, що буде вирішувати основні проблеми користувачів та не буде мати недоліків, які є у подібних систем, завдяки чому така система буде більш ефективною та повністю задовільняти як користувачів, що бажають стати у чергу, так і користувачів, що бажають створити чергу, наприклад до свого кафе, аби уникнути скупчення людей під час пандемії.

Це допоможе як користувачам, що бажають стати у чергу для отримання послуг, так і користувачам, що будуть надавати послуги, оскільки вони не будуть втрачати клієнтів, що були б готові заплатити їм за надані послуги, але в силу відсутності інформації про можливість надання їм цих послуг відмовляються від своїх бажань та потреб, замість чого йдуть до інших надавачів послуг, або зовсім від них відмовляються.

На теперішній момент є дуже важливим зберігання безпечної дистанції між людьми, оскільки розповсюдження хвороби під час пандемії коронавірусу є надзвичайно важливим.

Завдяки розробленому сервісу цю проблему вдається вирішити, оскільки відпадає необхідність створення живої черги, що призводило до того, що люди були вимушені сидіти під кабінетом лікаря аби потрапити до нього на прийом та отримати необхідну консультацію, внаслідок чого вдається запобігти великому скупченню людей у замкненому просторі.

Розроблений сервіс має бути максимально зручним у користуванні, для цього він повинен мати зрозумілий графічний інтерфейс, завдяки якому будь-яка людина зможе ним користуватися. Для цього знадобиться написати обрамлення для програмної частини використовуючи відповідні фреймворки.

Також даний сервіс повинен зберігати дані на сервері, а не на клієнті для того, щоб будь-який користувач у будь-який час міг отримати потрібну йому

інформацію. Цю проблему сервіс буде вирішувати за допомогою бази даних (далі — БД), яка буде надавати усю відповідну інформацію.

Для того щоб уникнути непередбачуваних та неочікуваних колізій усі запити до БД мають бути синхронізованими, а саме: якщо два користувача одночасно забажають стати у чергу система повинна додати до черги обох користувачів, а не тільки останнього.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ЕЛЕКТРОННОЇ ЧЕРГИ ТА ОБҐРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ

1.1 Поняття електронної черги

Електронна черга — це такий програмно-апаратний комплекс, який дозволяє дистанційно ставати у чергу у деяке місце на деяку подію завдяки відповідному сервісу за допомогою телефону, ноутбуку, ПК, годиннику, планшету, або будь-якому іншому подібному пристрою з будь-якого місця, де є інтернет надаючи можливість усім її користувачам точно спланувати час відвідування тієї чи іншої установи та не витрачати час на непотрібні дії.

1.2 Приклади застосування електронної черги

Найпростіший приклад, який можна привести: замість того, щоб приходити до поліклініки, та сидіти під кабінетом лікаря, контактувати з хворими та гаяти час доки лікар приймає інших пацієнтів ви можете стати у чергу дистанційно завдяки вашому девайсу та прийти у зазначений час паралельно займаючись своїми справами.

Ще одним актуальним під час часткового карантину прикладом використання сервісу може слугувати використання у кафе, барах та ресторанах, де на теперішній момент в приміщенні може бути присутня обмежена кількість людей, яка може легко мінятися онлайн за декілька секунд, що дозволить усім бажаючим відвідати якийсь заклад подивитися чи є вільні місця у їх улюбленому, або запланованому місці і забронювати їх, якщо вони є, стати у чергу, якщо їх

немає, або взагалі піти у інше місце, де будуть вільні місця, якщо у них немає часу чекати.

Даний сервіс допоможе набагато зручніше виконати поставлену задачу, а саме: знайти куди піти ввечері, якщо ви ніякі місця заздалегідь не бронювали, оскільки коли всі бажаючі відвідати заклад будуть дзвонити на рецепцію, де по телефону будуть хотіти отримани всю бажану інформацію виникне проблема, при якій буде невистачати операторів, щоб обробити всі запити, у даному випадку дзвінки, в той час, як даний сервіс може дозволити одночасно всім бажаючим дізнатися потрібну інформацію.

Також потрібно зазначити, що підтримання актуальної інформацію про кількість вільних місць зі сторони закладу у сервісі набагато простіше ніж найм оператору рецепції, що буде безперестанно обробляти запити клієнтів, що надходять телефонним режимом.

1.3 Аналіз сучасних засобів електронної черги

На теперішній момент існує декілька систем, що використовують електронні черги, такими є ДМСУ та система helsi. Вони дозволяють не виходячи з дому стати у чергу, що економить користувачам сили, час та нерви. Такі сервіси значно спрощують життя та надають можливість робити більшу кількість справ за одиницю часу [2].

Система helsi дозволяє Вам обирати свого лікаря, подивитися його розклад, щоб знати коли він буде вільний та стати до нього на прийом у чергу.

Для ДМСУ обрали інший підхід, оскільки там всі робітники виконують одну й ту саму функцію, а саме: видають паспорти. У даній системі користувачу

пропонується стати у чергу, в якій система автоматично розраховує очікуваний час перебування у черзі в залежності від кількості людей у цій черзі та надає можливі дати на вибір користувачу.

Також потрібно проаналізувати математичні методи для розрахунку очікування у черзі, та обрати остаточний метод, який буде використовуватися, якщо замовнику такий функціонал підходить та потрібен.

Дані функції можна запрограмувати за допомогою СМО, оскільки це простий, зрозумілий та потужний математичний апарат.

1.4 Постановка задачі дослідження

На основі проведеного аналізу виявлено мету та вимоги до магістерської дисертації.

Метою є підвищення ефективності програмно-апаратних засобів сервісу електронної черги за рахунок зменшення часу очікування у черзі та реалізації нових функціональних можливостей дистанційної взаємодії із засобами цього сервісу.

При розробці відповідного забезпечення потрібно розв’язати наступні завдання:

а) провести порівняння аналізу існуючих методів підрахунку часу очікування у черзі;

б) вибрати та адаптувати існуючий метод для підрахунку часу очікування у черзі;

в) розробити ПЗ;

г) протестувати розроблений автоматизований сервіс.

Реалізований сервіс має задовольняти такі вимоги:

а) своєчасно відпрацьовувати по запитам користувачів;

б) має проходити без падінь повністю димне тестування та проходити щонайменше 80% регресійного.

Щоб бути зручним та корисним він має давати змогу стати у чергу віддалено, подивитися своє місце у черзі, додати можливість ставати у чергу кудись, якщо її досі немає, вийти з черги та багато іншого.

Черга у нашому розумінні має реалізовувати систему – перший прийшов, перший вийшов, але по бажанню замовника це можна змінити. Цей алгоритм є основним при роботі з чергою.

Сервіс має давати змогу записатися у чергу, виписатися із черги, подивитися своє місце у черзі та дізнатися приблизний час очікування у черзі.

Якщо людина не приходить вчасно, то можна або перенести її місце у черзі на одну позицію назад, або просто видалити її із черги, як, наприклад і робиться у системі ДМСУ, але знову ж таки, це все має обговорюватися із замовником до початку виконання робіт із програмування та реалізації сервісу електронної черги.

Висновки до розділу 1

На теперішній час є неможливим запис у чергу до будь-якої установи самотужки, особливо якщо мова йде про звичайну районну поліклініку, а не приватну, де роблять усе можливе, аби клієнти залишилися задоволеними. У таких місцях окрім їх власних імплементацій електронних черг можуть бути і виокремлені окремі працівники, що будуть слідкувати за такими чергами, хоча вони можуть працювати й без втручання операторів.

Існуючі системи очевидно є закритими, вузьконаправленими, можуть містити надлишкову інформацію та, навіть, порушувати конфіденційність та анонімність користувачів не виконуючи правила GDPR, або навіть крадучи інформацію з пристроїв користувачів їх сервісів.

Окрім викрадення даних такі сервіси можуть також у фоновому режимі виконувати різноманітні обчислювальні операції що тим самим дозволяє їм безкоштовно користуватися розрахунковими можливостями вашого девайсу, що призведе до сповільнення його роботи, збільшення їх часу роботи, що призведе до їх скорішого виходу із ладу.

Окрім викрадення даних такі сервіси можуть також у фоновому режимі виконувати різноманітні обчислювальні операції що тим самим дозволяє їм безкоштовно користуватися розрахунковими можливостями вашого девайсу, що призведе до сповільнення його роботи, збільшення їх часу роботи, що призведе до їх скорішого виходу із ладу.

Також треба наголосити, що всі системи електронних черг наразі орієнтовані на людей, що надають послуги, а не на тих, що їх отримують. Розроблений сервіс же навпаки робить все можливе, аби обидві сторони купівлі-продажу послуг залишилися максимально задоволеними.

Підсумовуючи надану вище інформацію можна коротко описати

необхідний сервіс, який має: бути легким у використанні, надійним, самодостатнім, тобто таким, що не потребує, але може мати, оператора, що слідкує за порядком черги зі сторони організатора події, швидким, цілісним, зрозумілим та таким, до якого буде доступ завдяки мережі інтернет за допомогою браузера, або спеціального додатку.

РОЗДІЛ 2. ТЕОРЕТИЧНЕ ПІДҐРУНТЯ РЕАЛІЗАЦІЇ КЕРУВАННЯ ЧЕРГОЮ ТА ПРАКТИЧНОЇ РЕАЛІЗАЦІЇ ЗА ДОПОМОГОЮ ОБРАНИХ ТЕХНОЛОГІЙ

2.1 Особливості технології REST

REST — підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів [3].

В основі REST закладено принципи функціонування Всесвітньої павутини і, зокрема, можливості HTTP. Дані повинні передаватися у вигляді невеликої кількості стандартних форматів (наприклад, HTML, XML, JSON). Будь-який REST протокол (HTTP в тому числі) повинен підтримувати кешування, не повинен залежати від мережевого прошарку, не повинен зберігати інформації про стан між парами «запит-відповідь». Стверджується, що такий підхід забезпечує масштабованість системи і дозволяє їй еволюціонувати з новими вимогами.

REST — це архітектурний стиль для розподілених гіпертекстових систем [4]. Насамперед це клієнт-серверна архітектура. Її обмеження вимагає розділення відповідальності між компонентами, які займаються зберіганням та оновленням даних (сервером), і тими компонентами, які займаються відображенням даних на інтерфейсі користувача та реагування на дії з цим інтерфейсом (клієнтом). Таке розділення дозволяє компонентам еволюціонувати незалежно. Схема робот изображена на рис.1 [5].

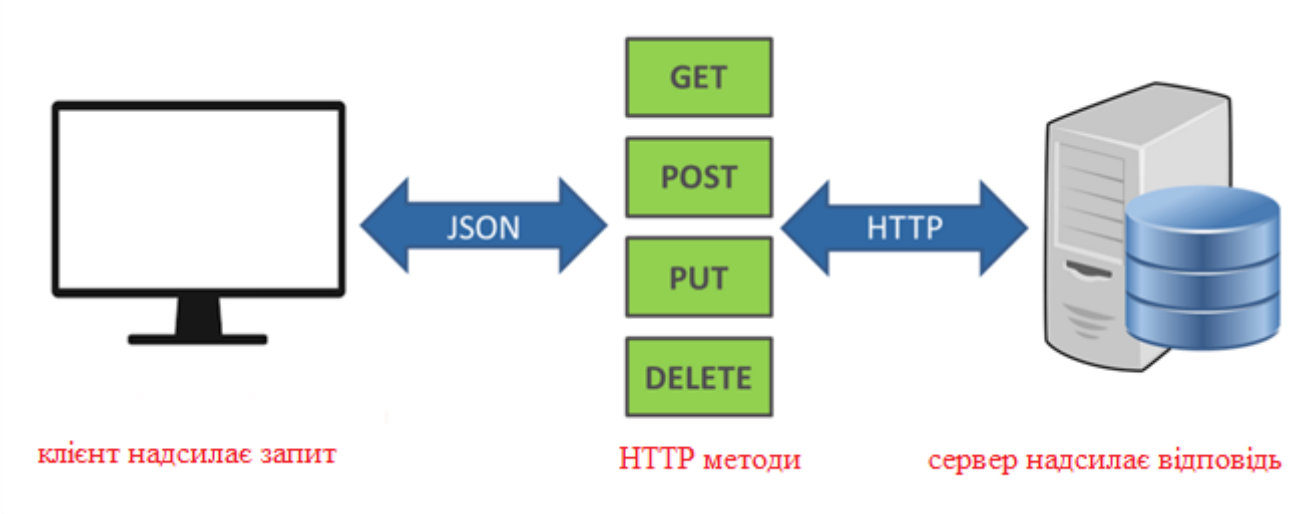

Рисунок 1 — Схема роботи REST запитів

У загальному випадку REST є дуже простим інтерфейсом управління інформацією без використання якихось додаткових внутрішніх прошарків. Кожна одиниця інформації однозначно визначається глобальним ідентифікатором, таким як URL. Кожна URL в свою чергу має строго заданий формат [4].

Відсутність додаткових внутрішніх прошарків означає передачу даних в тому ж вигляді, що і самі дані. Тобто дана архітектура не загортає дані в XML, як це робить SOAP і XML-RPC, не використовуємо AMF, як це робить Flash і т.д. Просто віддаємо самі дані.

Кожна одиниця інформації однозначно визначається URL - це значить, що URL по суті є первинним ключем для одиниці даних. Тобто наприклад сьомий стелаж у меблевому магазині матиме вигляд — /rack/7, а друга полиця в цьому стелажі — /rack/7/shelf/2. Звідси і виходить строго заданий формат. Причому абсолютно не має значення, в якому форматі знаходяться дані за адресою /rack/7/shelf/2 - це може бути і HTML, і відсканована копія у вигляді jpeg файлу, і документ Microsoft Word. Архітектура REST API представлена на рис. 2 [6].

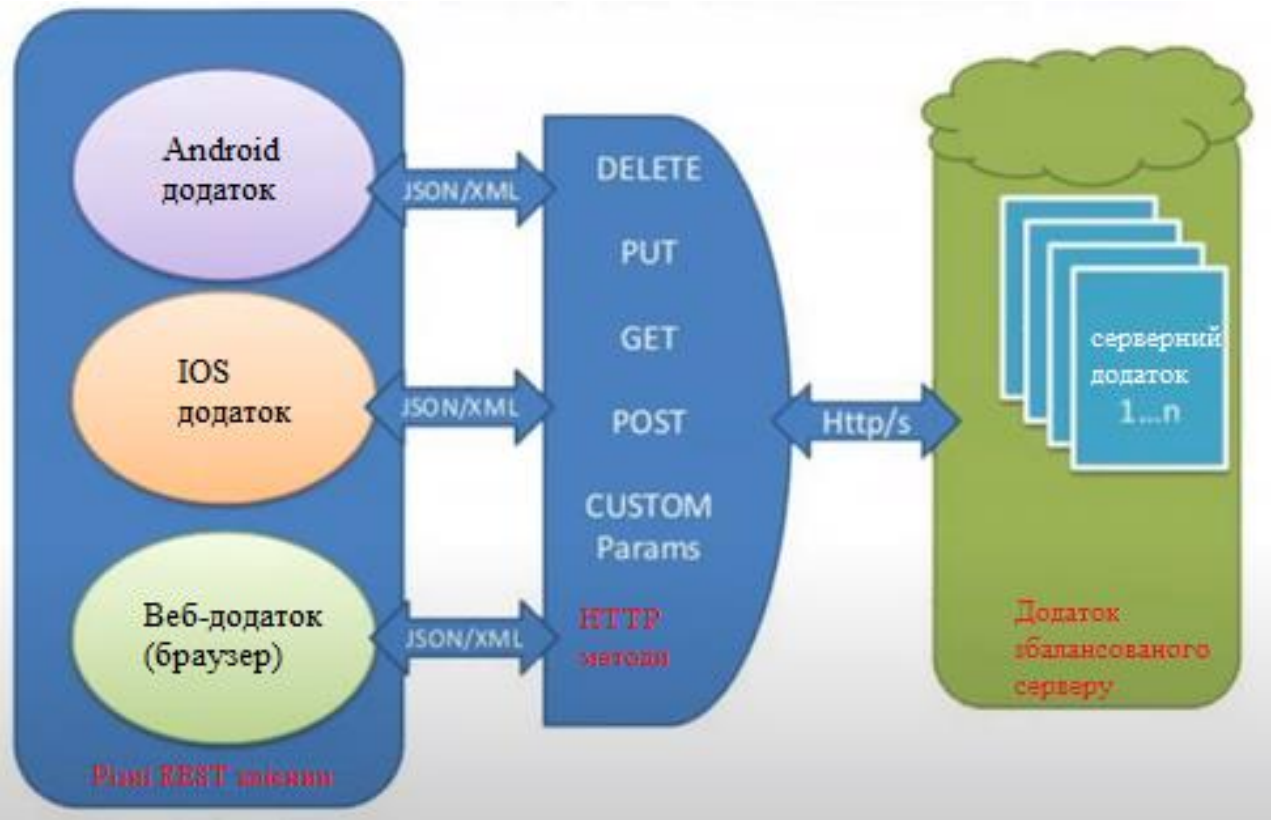

Рисунок 2 — Архітектура REST API

Як відбувається управління інформацією сервісу - це цілком і повністю ґрунтується на протоколі передачі даних. Найбільш поширений протокол звичайно ж HTTP або HTTPS. Так ось, для HTTP дію над даними задається за допомогою методів: GET (отримати), POST (додати, оновити, видалити), PUT (оновити), DELETE (видалити). Таким чином, дії CRUD можуть виконуватися як з усіма 4-ма методами, так і тільки за допомогою GET і POST.

Ось як це буде виглядати на прикладі:

GET /rack/ — отримати список всіх стелажів

GET /rack/7/ — отримати стелаж номер 7

POST /rack/ — додати стелаж (дані в тілі запиту)

PUT /rack/7 — оновити сьомий стелаж (дані в тілі запиту)

DELETE /rack/7 — видалити сьомий стелаж

Існують так звані REST-Patterns, які розрізняються зв'язуванням HTTP-методів з тим, що вони роблять. Зокрема, різні патерни по-різному розглядають POST і PUT. Однак, PUT призначений для створення, реплейса або апдейта, для POST це не визначено. Тому даний приклад буде правильним і в такому вигляді, і у вигляді якщо замінити PUT і DELETE на POST.

Взагалі, POST може використовуватися одночасно для всіх дій зміни:

POST /rack/ — додати стелаж (дані в тілі запиту)

POST /rack/7/ — оновити сьомий стелаж (дані в тілі запиту)

POST /rack/7/ — видалити сьомий стелаж (тіло запиту порожнє)

Це дозволяє іноді обходити неприємні моменти, пов'язані з неприйняттям PUT і DELETE.

Як відомо, web-сервіс - це додаток працює в World Wide Web і доступ до якого надається по HTTP-протоколу, а обмін інформацією йде за допомогою формату XML. Отже, формат даних переданих в тілі запиту буде завжди XML.

Як видно, в архітектура REST дуже проста в плані використання. По виду прийшов запиту відразу можна визначити, що він робить, не розбираючись в форматах (на відміну від SOAP, XML-RPC). Дані передаються без застосування додаткових шарів, тому REST вважається менш ресурсномістких, оскільки не треба розбирати запит щоб зрозуміти що він повинен зробити і не треба переводити дані з одного формату в інший.

Найбільша перевага таких сервісів в тому, що з ними працювати може будь-яка система, сайт, flash, програма та інші. Так як методи розбору XML і виконання запитів HTTP присутні майже всюди.

Архітектура REST дозволяє серйозно спростити цю задачу.

2.2 Переваги застосування мікрофреймворку Flask для вирішення поставленої задачі

Flask — мікрофреймворк для веб-додатків, створений з використанням Python [7]. Його основу складає інструментарій Werkzeug та рушій шаблонів Jinja2.

Flask називається мікрофреймворком, оскільки він не вимагає спеціальних засобів чи бібліотек. У ньому відсутній рівень абстракції для роботи з базою даних, перевірки форм або інші компоненти, які надають широковживані функції за допомогою сторонніх бібліотек. Однак, Flask має підтримку розширень, які забезпечують додаткові властивості таким чином, наче вони були доступні у Flask із самого початку. Існують розширення для встановлення об'єктно-реляційних зв'язків, перевірки форм, контролю процесу завантаження, підтримки різноманітних відкритих технологій аутентифікації та декількох поширених засобів для фреймворку. Розширення оновлюються частіше аніж базовий код.

«Мікро» у фреймворці відноситься не лише до простоти та невеликого розміру БД, але це також може означати той факт, що він не пропонує вам багато проектних рішень. Тим не менш, для нас термін «мікро» не означає, що вся реалізація повинна вписатися в один файл.

Одним із проектних рішень у Flask є те, що прості завдання повинні бути простими; вони не повинні займати багато коду і це не повинно обмежувати користувачів.

У Flask багато речей попередньо сконфігуровані на основі загальної базової конфігурації. Наприклад, шаблони та статичні файли зберігаються у підкаталогах в межах вихідного дерева. Ви також можете змінити це, але зазвичай це не потрібно.

Основна причина того, що Flask називається «мікрофреймворком» - це ідея зберегти ядро простим, але розширюваним. У ньому нема абстрактного рівня БД, нема валідації форм або усього того, що вже є в інших бібліотеках. Однак Flask підтримує розширення, яке може додати необхідну функціональність і впроваджує їх так, ніби вони вже були вбудовані спочатку. На даний момент вже є розширення: перевірка форм, підтримка завантаження файлів та багато інших.

Ваш безпечний веб-додаток можна зламати різними способами, так як веб-програмування - це небезпечне заняття. Ви дозволяєте користувачам залишати інформацію на сервері, отже можна знайти спосіб зламати ваше веб-додаток. Flask захищає вас від найбільш поширених і відомих способів злому, такі як XSS. До тих пір, поки ви самі свідомо не відзначаєте небезпечний HTML як безпечний, Flask і шаблонізатор Jinja2 захищають вас, але все одно можуть знайтися способи зламати ваш сайт. Схема роботи XXS атаки представлена на рис.3 [8].

Рисунок 3 — Схема роботи XXS атаки

Основні властивості через які обрано даний фреймворк це:

а) наявність серверної підтримки;

б) управління RESTful запитами;

в) масштабованість;

г) простота використання.

2.3 Розгляд баз даних

2.3.1 Огляд бази даних PostgreSQL

PostgreSQL — об'єктно-реляційна СКБД. Є альтернативою як комерційним СКБД (Oracle Database, Microsoft SQL Server, IBM DB2 та інші), так і СКБД з відкритим кодом (MySQL, Firebird, SQLite) [9].

Порівняно з іншими проектами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

Сервер PostgreSQL написаний на мові C.

Функції дозволяють виконувати деякий код безпосередньо сервером бази даних. Ці функції можуть бути написані на SQL, який має деякі примітивні програмні оператори, такі як галуження та цикли. Але гнучкішою буде функція написана на одній із мов програмування, з якими PostgreSQL може працювати. До таких мов належать:

а) вбудована мова, яка зветься PL/pgSQL, подібна до процедурної мови PL/SQL компанії Oracle.

б) мови розробки сценаріїв: PL/Perl, PL/Python, PL/Tcl, PL/Ruby, PL/sh.

в) класичні мови програмування C, C++, Java (за допомогою PL/Java).

г) функції можуть виконуватись із привілеями користувача, який її викликав, або із привілеями користувача, який її написав.

Тригери визначаються як функції, що ініціюються DML-операціями. Наприклад, операція INSERT може запускати тригер, що перевіряє доданий запис на відповідність певним умовам. Тригери можна писати різними мовами програмування. Вони пов'язані з визначеною таблицею. Множинні тригери виконуються в алфавітному порядку.

2.3.2 Огляд бази даних MongoBD

MongoDB — документо-орієнтована СКБД з відкритим вихідним кодом, яка не потребує опису схеми таблиць. MongoDB займає нішу між швидкими і масштабованими системами, що оперують даними у форматі ключ/значення, і реляційними СКБД, функціональними і зручними у формуванні запитів [10].

Код MongoDB написаний на мові C++ і поширюється в рамках ліцензії AGPLv3.

MongoDB підтримує зберігання документів в JSON-подібному форматі, має досить гнучку мову для формування запитів, може створювати індекси для різних збережених атрибутів, ефективно забезпечує зберігання великих бінарних об'єктів, підтримує журналювання операцій зі зміни і додавання даних в БД, може працювати відповідно до парадигми Map/Reduce, підтримує реплікацію і побудову відмовостійких конфігурацій.

У MongoDB є вбудовані засоби із забезпечення шардінгу (розподіл набору даних по серверах на основі певного ключа), комбінуючи який з реплікацією

даних можна побудувати горизонтально масштабований кластер зберігання, в якому відсутня єдина точка відмови (збій будь-якого вузла не позначається на роботі БД), підтримується автоматичне відновлення після збою і перенесення навантаження з вузла, який вийшов з ладу. Розширення кластера або перетворення одного сервера на кластер проводиться без зупинки роботи БД простим додаванням нових машин.

Основні можливості MongoDB:

а) документо-орієнтоване сховище (проста та потужна JSON-подібна схема даних);
б) досить гнучка мова для формування запитів;
в) динамічні запити;
г) повна підтримка індексів;
д) профілювання запитів;
е) швидкі оновлення «на місці»;
ж) ефективне зберігання бінарних даних великих обсягів, наприклад, фото та відео;
з) логування операцій, що модифікують дані в БД;
и) підтримка відмовостійкості і масштабованості: асинхронна реплікація, набір реплік і шардінг;
к) може працювати відповідно до парадигми MapReduce.

2.3.3 Огляд бази даних Cassandra

Apache Cassandra — вільна та відкрита розподілена з широким стовпчиком noSQL система керування базами даних, яка створена для роботи з

високомасштабованими і надійними сховищами величезних масивів даних. Cassandra надає надійну підтримку кластерів, що охоплюють численні датацентри та забезпечує високу доступність даних та працює без точкових відмов з асинхронною нецентралізованою реплікацією даних, що дозволяє для всіх користувачів виконувати операції з низькою затримкою [11].

СКБД Cassandra написана на мові Java і об'єднує в собі повністю розподілену hash-систему Dynamo, що забезпечує практично лінійну масштабованість при збільшенні обсягу даних. Cassandra використовує модель зберігання даних на базі сімейства стовпців (ColumnFamily), що відрізняється від систем подібних до memcachedb, які зберігають дані лише у зв'язці ключ/значення, можливістю організувати зберігання хешей з кількома рівнями вкладеності. Cassandra відноситься до категорії сховищ підвищено стійких до збоїв: поміщені в БД дані автоматично реплікуються на кілька вузлів розподіленої мережі або навіть рівномірно розподіляються до декількох дата-центрів. При збої вузла, його функції на льоту підхоплюються іншими вузлами. Додавання нових вузлів у кластер і оновлення версії Cassandra виробляється на льоту, без додаткового ручного втручання і переконфігурації інших вузлів.

Для спрощення взаємодії з БД підтримується мова формування структурованих запитів CQL (Cassandra Query Language), яка на перший погляд нагадує SQL, але істотно урізана в функціональності. Наприклад, можна виконувати тільки найпростіші запити SELECT з вибіркою за певною умовою, але без підтримки сортування та групування. Додавання та оновлення даних здійснюється через вирази UPDATE або INSERT. Вони схожі за поведінкою і додають новий запис якщо запис відсутній або оновлюють існуючий. З можливостей можна відзначити підтримку просторів імен і сімейств стовпців, створення індексів через вираз "CREATE INDEX". Драйвери з підтримкою SQL підготовлені для мов Python, Java (JDBC/DBAPI2) і JavaScript (Node.js).

2.3.4 Огляд бази даних Oracle

Oracle Database — об'єктно-реляційна СКБД від Oracle Corporation [12].

Особливості [13]:

а) підтримання декількох версій даних для управління паралельними транзакціями;

б) секціонування;

в) автономні транзакції;

г) автоматичне керування зберіганням файлів БД;

д) набір інструментів, призначених для управління і моніторингу СКБД Oracle і серверів, на яких вони встановлені;

е) пакети;

ж) підтримка послідовностей;

з) аналітичні функції в SQL;

и) profile manager;

к) Oracle Label Security;

л) Streams;

м) Advanced Queuing;

н) Flashback Query;

о) RAC (англ. Real Application Clusters);

п) RAT (Real Application Testing) - дозволяє значно знизити витрати на випробування нової конфігурації програмного або апаратного забезпечення, так як здатна точно відтворити на ній навантаження робочого сервера;

р) Data Guard - технологія, що дозволяє створити резервний сервер, який може працювати в парі з основним, знижуючи навантаження на нього, і який може автоматично замінити основний сервер в разі його відмови або планового

відключення (є варіант з постійною доступністю резервного сервера для читання - Active Data Guard);

с) Total Recall - дає можливість розвантажити базу даних від застарілої, не використовуваної інформації, зберігаючи при цьому можливість доступу до неї, так що для користувача бази даних це зміна залишається непоміченим;

т) об'єктні типи (в сенсі об'єктно-орієнтованого підходу);

у) Automatic Database Diagnostic Monitoring - автоматичний моніторинг і діагностика баз для виявлення проблем продуктивності та, можливо, автоматичної корекції (якщо така визначена адміністратором);

ф) підказки для зміни плану виконання запиту.

Oracle зазвичай використовується для запуску обробки онлайн-транзакцій, зберігання даних та змішаних навантажень БД. Вона може бути доступна декількома постачальниками послуг як попередньо, так і в хмарі або як гібридна хмарна установка. Вона може працювати на сторонніх серверах, а також на апаратному забезпеченні Oracle. Ексклюзивно для клієнтів Cloud, Oracle пропонує Oracle Autonomous Database, що забезпечує повністю автоматизовані процедури роботи.

Продукти баз даних Oracle все частіше конкурують з такими реляційними та нереляційними системами баз даних з відкритим кодом, як PostgreSQL, MongoDB, Couchbase, Neo4j та іншими. Oracle придбав Innobase, постачальника кодової бази InnoDB для MySQL, частково, щоб краще конкурувати з альтернативами з відкритим кодом, і придбав Sun Microsystems, власника MySQL для інтегрування додаткових функцій у своє ПЗ, що раніше були впливовим аргументом, аби використовувати інших компаній [14].

Завдяки цьому функціонал, що надає компанія Oracle задовольняє майже усіх користувачів своєю повнотою та зручністю використання.

2.4 Обґрунтування вибору архітектури

Основні відмінності SQL та NoSQL [15]:

а) Схема

У SQL БД всі дані мають чітку схему даних притримуючись якої описуються всі дані, що вносяться в таблицю. У таблиці 1 показано приклад таблиці SQL схеми, що представляє собою таблицю що містить інформацію про товари, у якої всі є поля (колонки) одного типу та всі записи (рядки) мають одні й ті самі поля.

Таблиця 1 — структура SQL бази даних

Номер	Назва	Ціна	Опис товару
1	Книга	48,50	Книга про чьотирьох друзів, що відкрили кафе
2	Стілець	299,99	Білий дерев'яний стілець 400x400x700
3	Мікрофон	1350,00	Дуже якісний бездротовий мікрофон

В NoSQL БД записи можуть відрізнятися, тобто вони не мають загальної схеми.

Таблиця 2 — структура NoSQL бази даних

Номер: 1	Ім'я: Максим	Вік: 25	
Номер: 2	Ім'я: Олег		Місто: Харків
Номер: 3		Вік: 31	Місто: Київ

б) Відношення

SQL БД з легкістю підтримують відношення між таблицями за рахунок зовнішніх ключів. У таблиці 3 наведено приклад таблиці «Замовлення», всі записи якої складається з трьох полів: номера замовлення, номера користувача та номера продукту, що беруться з інших таблиць таких як «Користувачі» та «Продукти» відповідно.

Таблиця 3 — Відношення таблиць

Користувачі

Номер	Електронна пошта	Ім'я
1	max@test.com	Максим
2	antony@ukr.net	Антон
3	k_oleg@gmail.com	Олег

Замовлення

Номер	Номер користувача	Номер продукту
1	2	1
2	1	1
3	2	2

Продукти

Номер	Назва	Ціна
1	Книга	48,50
2	Стілець	299,99
3	Мікрофон	1350,00

«Номер користувача» та «Номер продукту» у таблиці «Замовлення» — це зовнішні ключі даної таблиці, що є головними ключами таблиць «Користувачі» та «Продукти» відповідно.

У NoSQL немає даного функціоналу, оскільки дані не мають чіткої схеми, то всі записи трактуються як нові, через що дані без потреби дублюються. Приклад наведено у таблиці 4.

Таблиця 4 — Приклад побудови відношень у NoSQL

Замовлення

{ Номер: 1, Користувач: {Номер: 2, Електронна пошта: "antony@ukr.net"}, Продукт: {Номер: 1, Ціна: 48,50}}
{ Номер: 2, Користувач: {Номер: 1, Електронна пошта: "max@test.com"}, Продукт: {Номер: 2, Ціна: 299,99}}
{ Номер: 3, Користувач: {Номер: 3}, Продукт: {Номер: 3}}

Користувачі

{ Номер: 1, Електронна пошта: "max@test.com", Ім'я: "Максим" }
{ Номер: 2, Електронна пошта: "antony@ukr.net", Ім'я: "Антон" }
{ Номер: 3, Електронна пошта: "k_oleg@gmail.com", Ім'я: "Олег" }

Користувачі

{ Номер: 1, Назва: "Книга", Ціна: 48,50 }
{ Номер: 2, Назва: "Стілець", Ціна: 299,99 }
{ Номер: 3, Назва: "Мікрофон", Ціна: 1350,00 }

Через це таблиці у SQL системі можуть бути атомарними та об'єднуватися в спільні таблиці запобігаючи дублюванню даних, в той час, як у NoSQL такий підхід неможливий.

в) Масштабованість

Масштабованість буває двох типів:

1) вертикальна;
2) горизонтальна.

Вертикальна масштабованість означає збільшення потужності серверу шляхом збільшення його продуктивності та обчислювальної потужності.

Горизонтальна масштабованість передбачає збільшення кількості серверів, на яких зберігається БД.

SQL системи дуже важко (майже неможливо) розширювати горизонтально за рахунок залежності таблиць одна від одної. Вертикальна масштабованість можлива.

У свою чергу через те, що дані у NoSQL системах не мають жорсткого зв'язку між собою такі системи можна розширювати як вертикально так и горизонтально. Типи масштабованості БД представлено на рис. 4 [15].

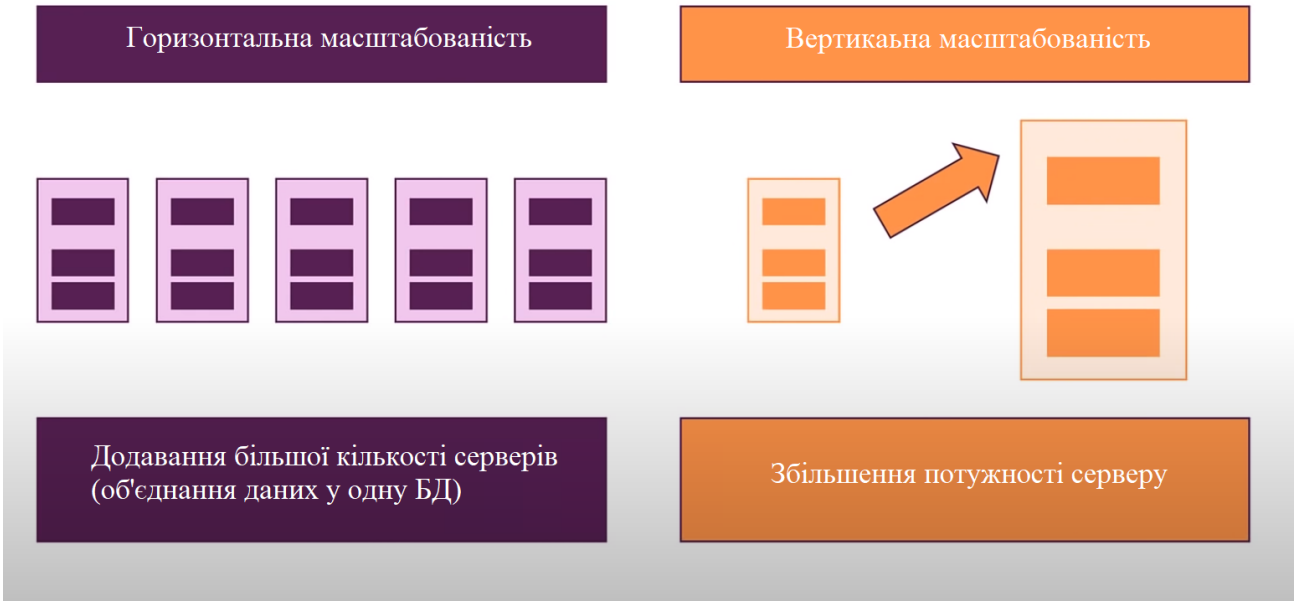

Рисунок 4 — Типи масштабованості баз даних

Беручи до уваги усі переваги та недоліки обох систем можемо зробити висновок що для сервісу електронної черги SQL система підходить краще, аніж NoSQL.

2.5 Проектування бази даних

Перед тим як створювати таблиці, форми та інші об’єкти, потрібно задати структуру бази даних. Добра структура бази даних є основою для створення адекватної вимогам, ефективної бази даних. Сам процес проектування бази даних

являє собою складний процес проектування відображення опису предметної області у схему внутрішньої моделі даних. Перебіг цього процесу є послідовністю більш простих процесів проектування менш складних відображень. Ця послідовність у процесі проектування весь час уточнюється, вдосконалюється таким чином, щоб були визначені об’єкти, їх властивості та зв’язки, які будуть потрібні майбутнім користувачам системи.

Нижче наведені основні етапи проектування БД:

а) визначення мети створення БД.

б) визначення таблиць, що їх повинна містити БД.

в) визначення необхідних у таблиці полів.

г) завдання індивідуального значення кожному полю.

ґ) визначення зв’язків між таблицями.

д) відновлення структури БД.

е) додавання даних і створення запитів, форм, звітів та інших об’єктів БД.

є) використання засобів аналізу в СКБД.

Для того, щоб робота з базою даних не приносила постійних труднощів при роботі з нею вона обов’язково має бути нормалізована.

Нормалізація схеми БД — покроковий процес розбиття однієї таблиці відповідно до алгоритму нормалізації на декілька відношень на базі функціональних залежностей [16].

Нормальна форма — властивість відношення в реляційній моделі даних, що характеризує його з точки зору надмірності, яка потенційно може призвести до логічно помилкових результатів вибірки або зміни даних. Нормальна форма визначається як сукупність вимог, яким має задовольняти відношення.

Таким чином, схема реляційної бази даних переходить у першу, другу, третю і так далі нормальні форми. Якщо відношення відповідає критеріям

нормальної форми n та всіх попередніх нормальних форм, тоді вважається, що це відношення знаходиться у нормальній формі рівня n.

Перша нормальна форма (1НФ, 1NF) утворює ґрунт для структурованої схеми бази даних:

1) кожна таблиця повинна мати основний ключ: мінімальний набір колонок, які ідентифікують запис;

2) уникнення повторень груп (категорії даних, що можуть зустрічатись різну кількість разів в різних записах) правильно визначаючи неключові атрибути;

3) атомарність: кожен атрибут повинен мати лише одне значення, а не множину значень.

Друга нормальна форма (2НФ, 2NF) вимагає аби дані, що зберігаються в таблицях із композитним ключем, не залежали лише від частини ключа:

1) схема бази даних повинна відповідати вимогам першої нормальної форми;

2) дані, що повторно з'являються в декількох рядках, виносяться в окремі таблиці.

Третя нормальна форма (3НФ, 3NF) вимагає аби дані в таблиці залежали винятково від основного ключа:

1) схема бази даних повинна відповідати всім вимогам другої нормальної форми;

2) будь-яке поле, що не залежить від основного ключа та від будь-якого іншого поля, має виноситись в окрему таблицю.

Відношення знаходиться в нормальній формі Бойса — Кодда (НФБК, NFBC) тоді і лише тоді, коли детермінант кожної функціональної залежності є потенційним ключем. Якщо це правило не виконується, то щоб привести вказане відношення до НФБК, його слід розділити на два відношення шляхом двох

операцій проекції на кожну функціональну залежність, детермінант якої не є потенційним ключем:

1) проекція без атрибутів залежної частини такої функціональної залежності;

2) проекція на всі атрибути цієї функціональної залежності.

Визначення НФБК не потребує жодних умов попередніх нормальних форм. Якщо проводити нормалізацію послідовно, то в переважній більшості випадків при досягненні 3НФ автоматично будуть задовольнятися вимоги НФБК. 3НФ не збігається з НФБК лише тоді, коли одночасно виконуються такі 3 умови:

1) відношення має 2 або більше потенційних ключів.

2) ці потенційні ключі складені (містять більш ніж один атрибут)

3) ці потенційні ключі перекриваються, тобто мають щонайменше один спільний атрибут.

Четверта нормальна форма (4НФ, 4NF) потребує аби в схемі баз даних не було нетривіальних багатозначних залежностей множин атрибутів від будь чого, окрім надмножини ключа-кандидата. Вважається, що таблиця знаходиться у 4НФ тоді і лише тоді, коли вона знаходиться в НФБК та багатозначні залежності є функціональними залежностями. Четверта нормальна форма усуває небажані структури даних — багатозначні залежності.

П'ята нормальна форма (5НФ, 5NF, PJ/NF) вимагає аби не було нетривіальних залежностей об'єднання, котрі б не витікали із обмежень ключів. Вважається, що таблиця в п'ятій нормальній формі тоді і лише тоді, коли вона знаходиться в 4НФ та кожна залежність об'єднання зумовлена її ключами-кандидатами.

Доменно-ключова нормальна форма вимагає, аби в схемі не було інших обмежень окрім ключів та доменів.

Таблиця знаходиться у шостій нормальній формі 6NF, якщо вона знаходиться у 5NF та задовольняє вимозі відсутності нетривіальних залежностей. Зазвичай 6NF ототожнюють з DKNF.

2.6 Візуалізація даних

Сервіс представляє собою сайт написаний мовою Python за допомогою фреймворку Flask, у якому валідація відбувається як на клієнті (за допомогою HTML), так і на сервері (за допомогою WTForms), а уся робота з інформацією відбувається за допомогою бази даних Oragle 11g. Усі виклики реалізовані за рахунок REST. Така структура спроектована за рахунок її простоти, надійності, зручності у користуванні та безпечності.

Мапа сайту представлена на рис. 5.

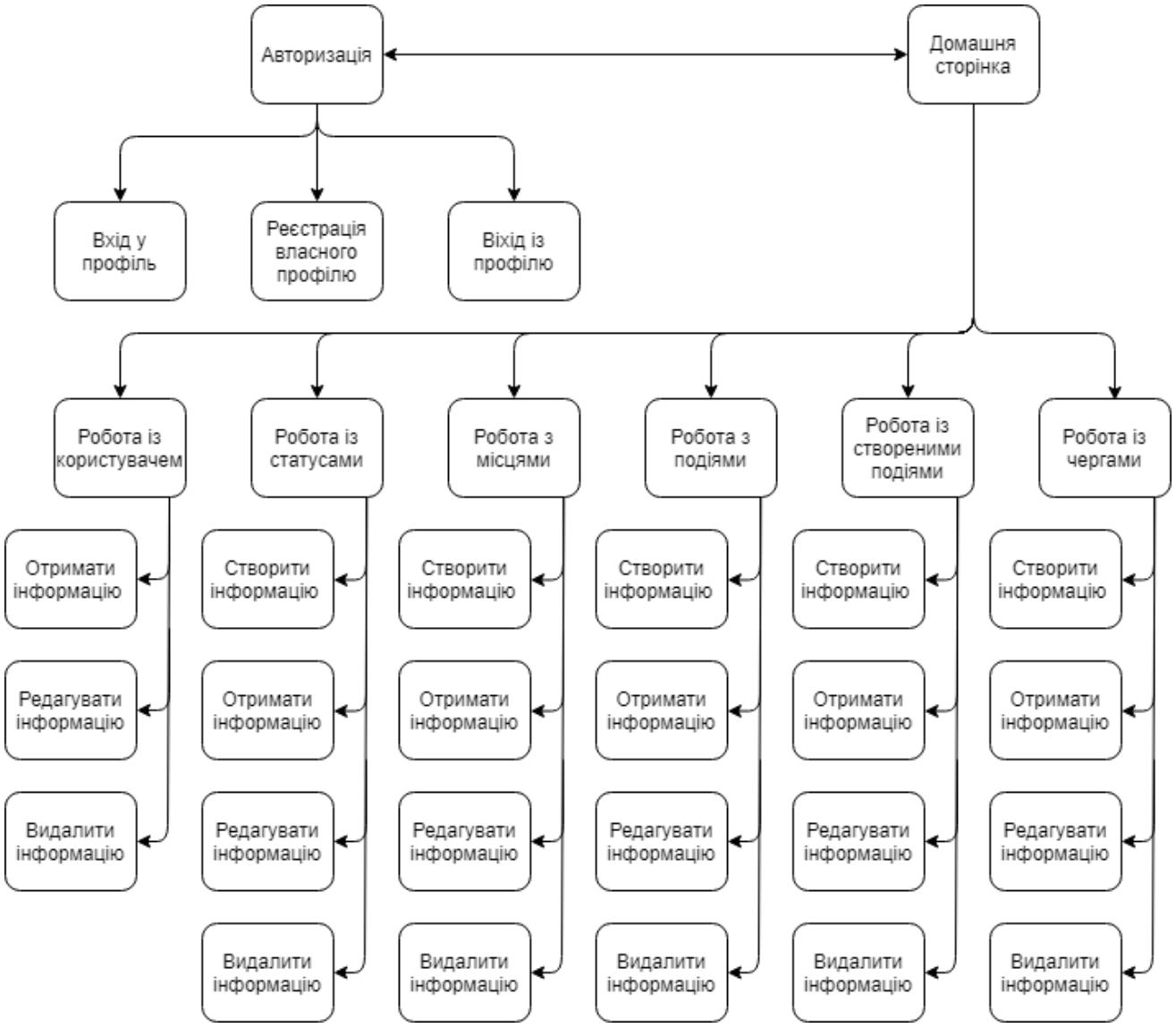

Рисунок 5 — Мапа сайту.

До множини сутностей, з якими буде налаштована робота, відносяться наступні:

а) користувачі — сутність, що дозволяє виконувати основні функції створення, зчитування, оновлення та видалення з користувачами;

б) статуси — сутність, що дозволяє виконувати основні функції створення, зчитування, оновлення та видалення із статусами. Представляє собою статус у черзі, а саме: стою у черзі, або уже не хочу і іду з неї;

в) місця — сутність, що дозволяє виконувати основні функції створення,

зчитування, оновлення та видалення з місцями;

г) події — сутність, що дозволяє виконувати основні функції створення, зчитування, оновлення та видалення з подіями. Подією може виступати будь-що, наприклад, похід у гори, на пікнік, візит до лікаря тощо;

д) створені події у місці — сутність, що дозволяє виконувати основні функції створення, зчитування, оновлення та видалення із створеними подіями у місці;

е) черги — сутність, що дозволяє виконувати основні функції створення, зчитування, оновлення та видалення з чергами;

2.6.1 Класи даних

До класів даних відносять наступні елементи:

а) Дані що необхідні для реєстрації користувача:

1) логін;

2) пароль;

3) електронна пошта.

б) Дані, які необхідні для авторизації користувача:

1) логін;

2) пароль.

в) Дані, які необхідні для створення події:

1) назва події.

г) Дані що необхідні для створення місця:

1) унікальний номер місця;

2) адреса;

3) номер приміщення;

4) розклад роботи.

д) Дані, які необхідні для створення статусу:

1) статус.

е) Дані що необхідні для створення заходу на який можна стати у чергу:

1) унікальний номер місця;

2) назва події;

3) дата створення заходу.

є) Дані що необхідні для черги:

1) статус;

2) логін;

3) унікальний номер місця;

4) назва події;

5) дата створення заходу;

6) дата створення черги.

Була створена діаграма компонентів, що відображає залежності між компонентами програмного забезпечення, включаючи компоненти вихідних кодів та компоненти, що можуть виконуватись. Зображено дії які може робити користувач у сервісі. Все це представлено у вигляді Use-Case діаграми на рис. 6.

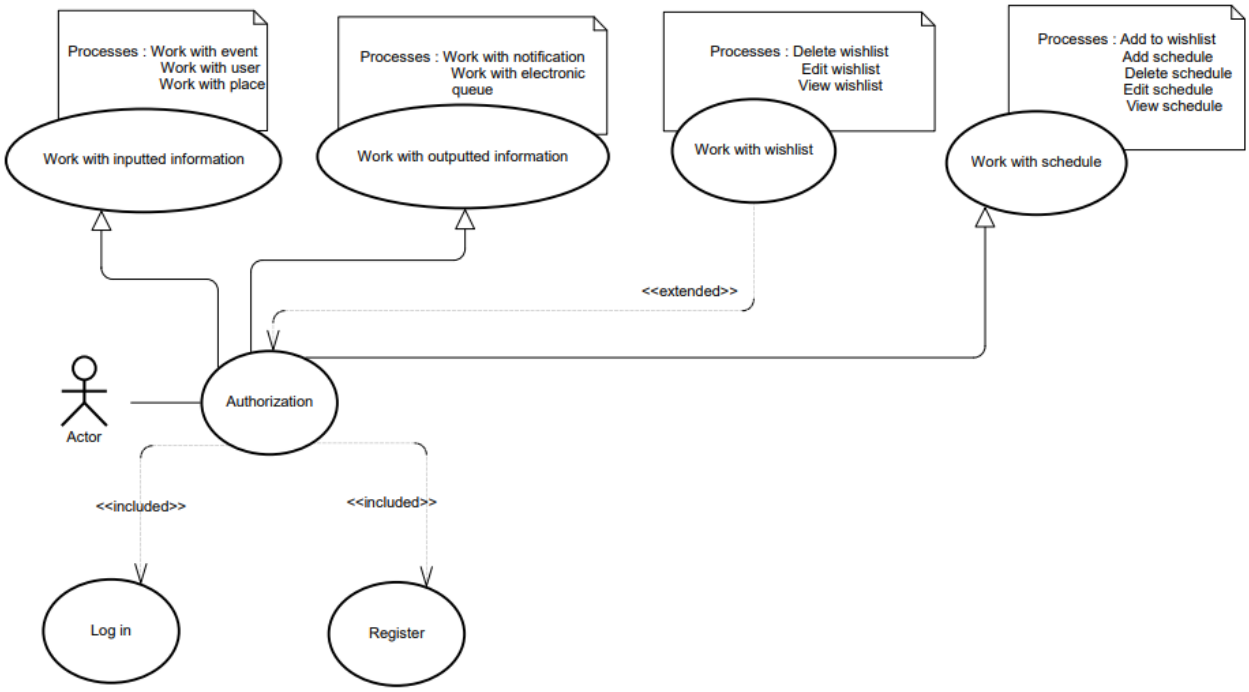

Рисунок 6 — Use-Case діаграма

Розроблено концептуальну (рис. 7), логічну (рис. 8), логічну оптимізовану (рис. 9) та фізичну (рис. 10) ERD діаграми.

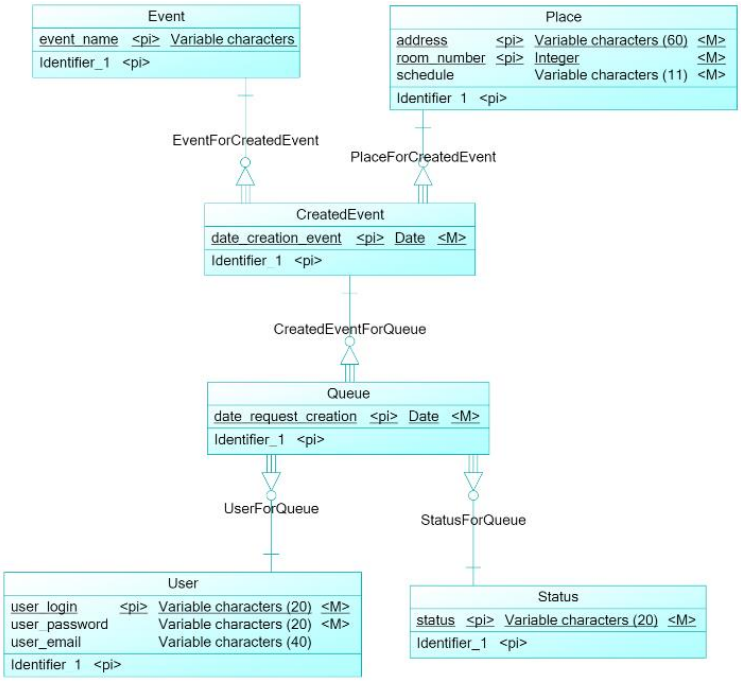

Рисунок 7 — Концептуальна діаграма

Логічні діаграми – описують склад, структуру, стан або поведінку елементів системи без прив'язки до конкретних мов або середовищ програмування, СКБД, технічних засобів і т. д. При розробці системи це забезпечує гнучкість у виборі і швидкий перехід з однієї програмно-апаратної платформи на іншу.

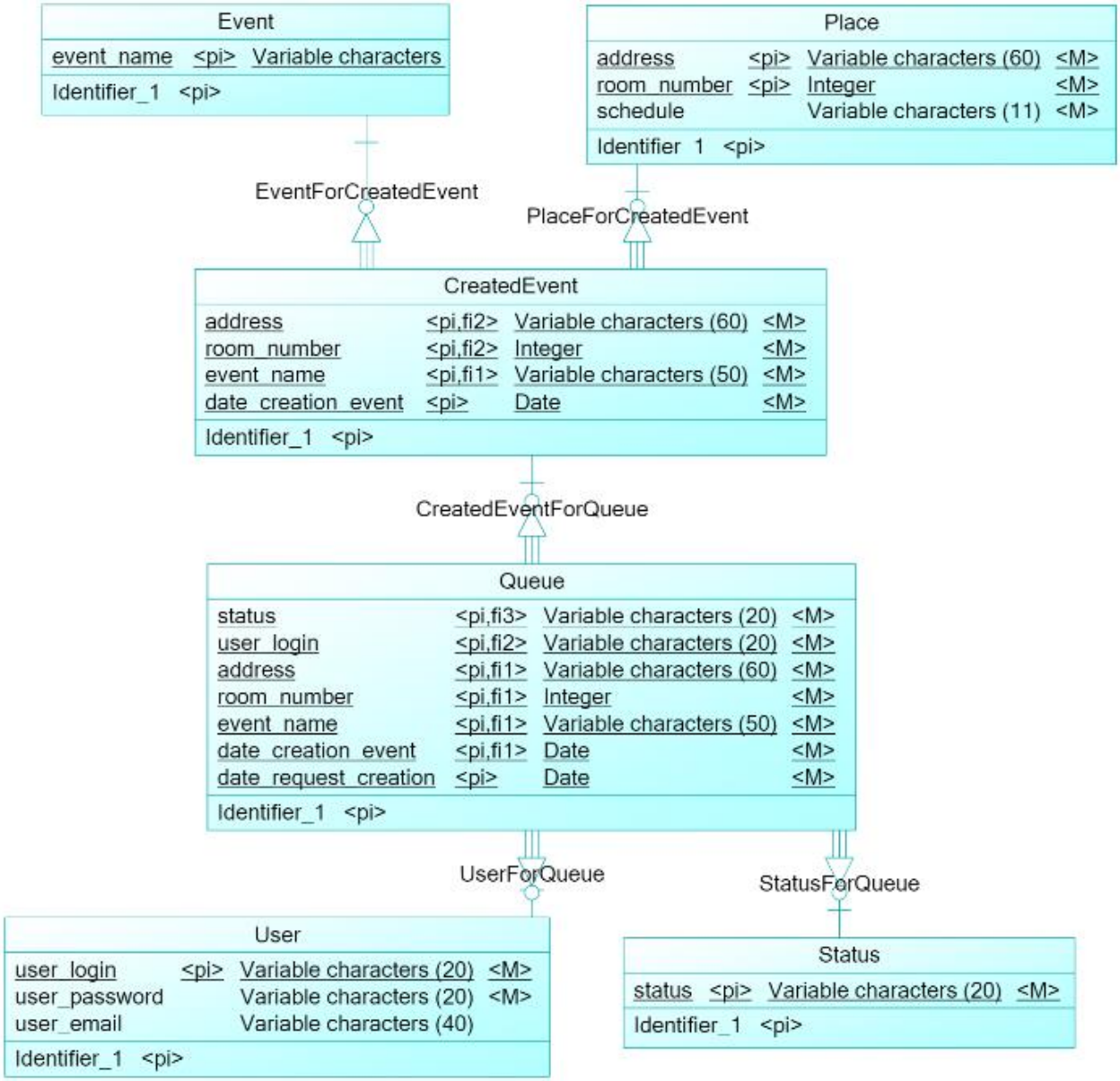

Рисунок 8 — Логічна діаграма

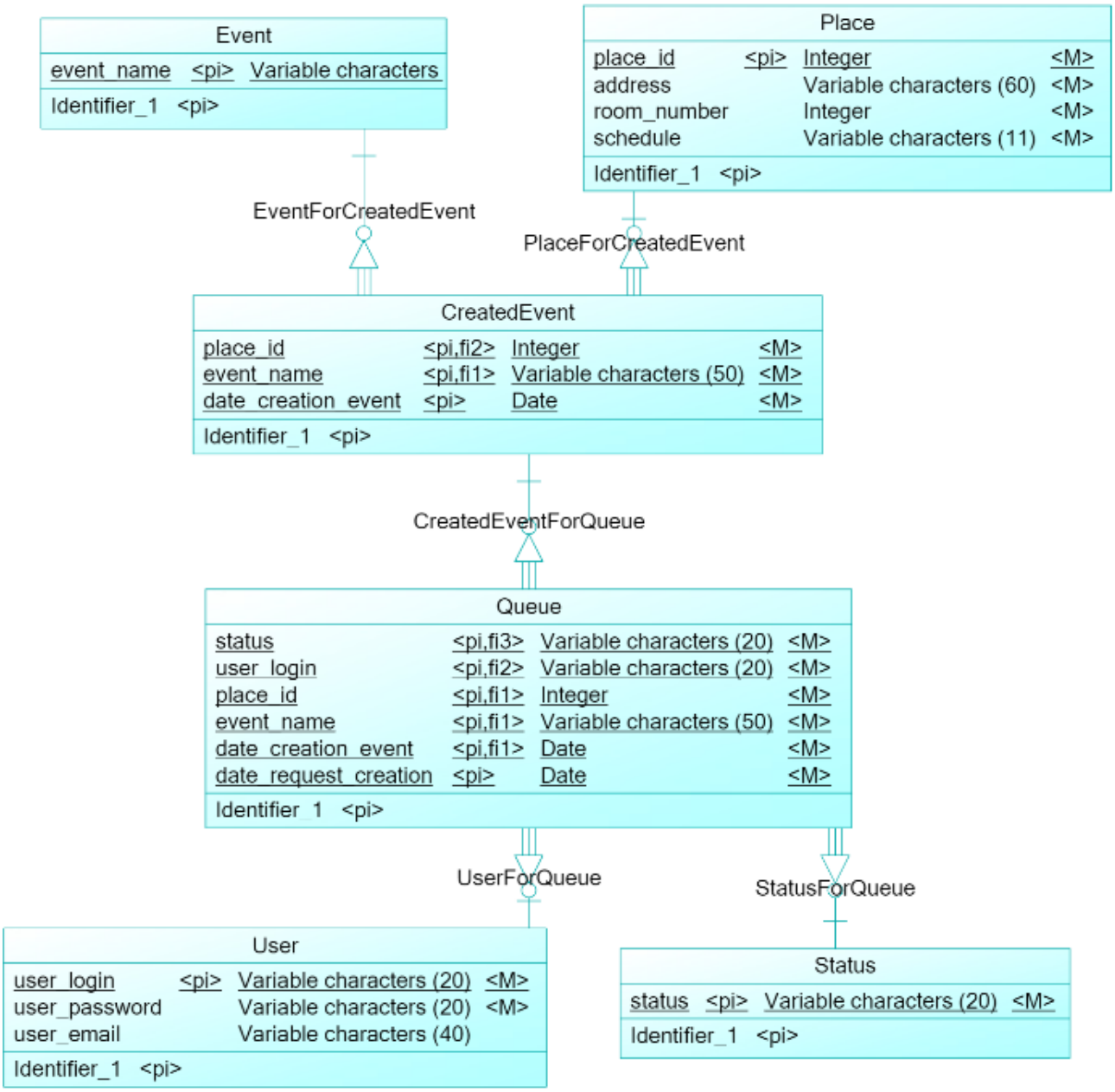

Рисунок 9 — Логічна оптимізована діаграма

Фізичні діаграми – описують елементи системи відповідно до прийнятої фізичної реалізації цих елементів (мов програмування, СКБД, пристроїв і т. д.).

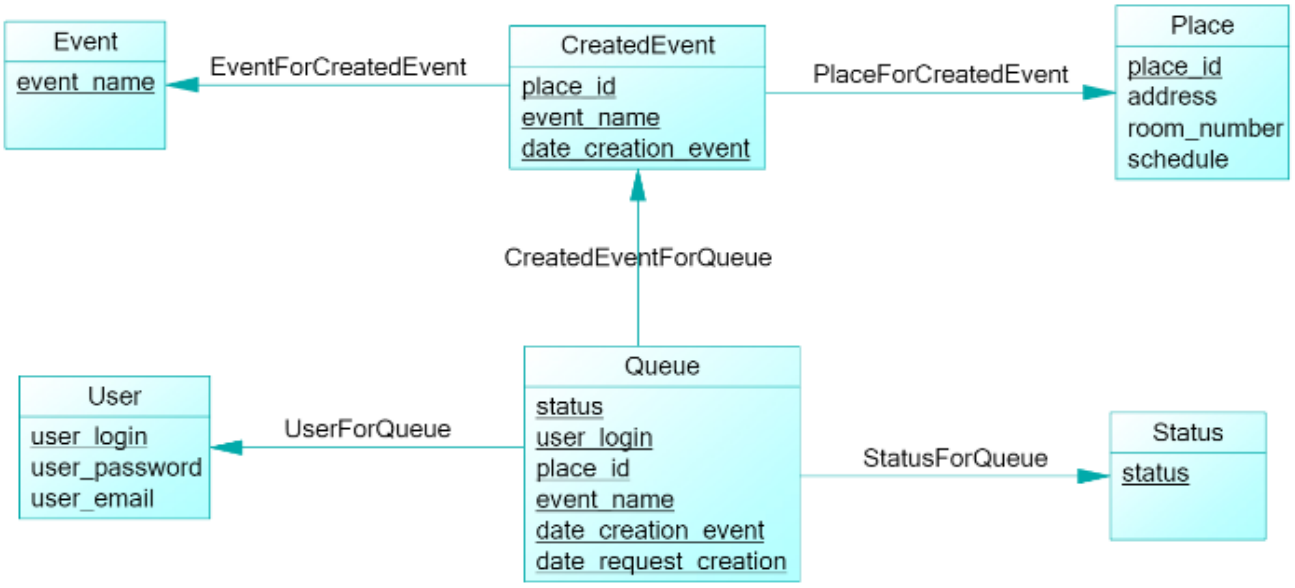

Рисунок 10 — Фізична діаграма

Для того щоб користувач не мав постійно вводити свої дані для при спробі входу в свій аккаунт сервісом передбачена робота з cookie та сесіями.

В cookie та сесію кладуться логін та пароль від аккаунту, які зберігаються на клієнті та сервері відповідно. Робота із сесією є більш швидкою, оскільки серверу не потрібно посилати запит для отримання даних та не потрібно чекати, доки клієнт відповість. Але так як місце на сервері потрібно економити, то й подібні дані зберігають на клієнті. Срок дії cookie встановлений 90 днів, але це легко змінити, за вимогою замовника. Блок-схема, що показує роботу даного блоку показана на рис. 11.

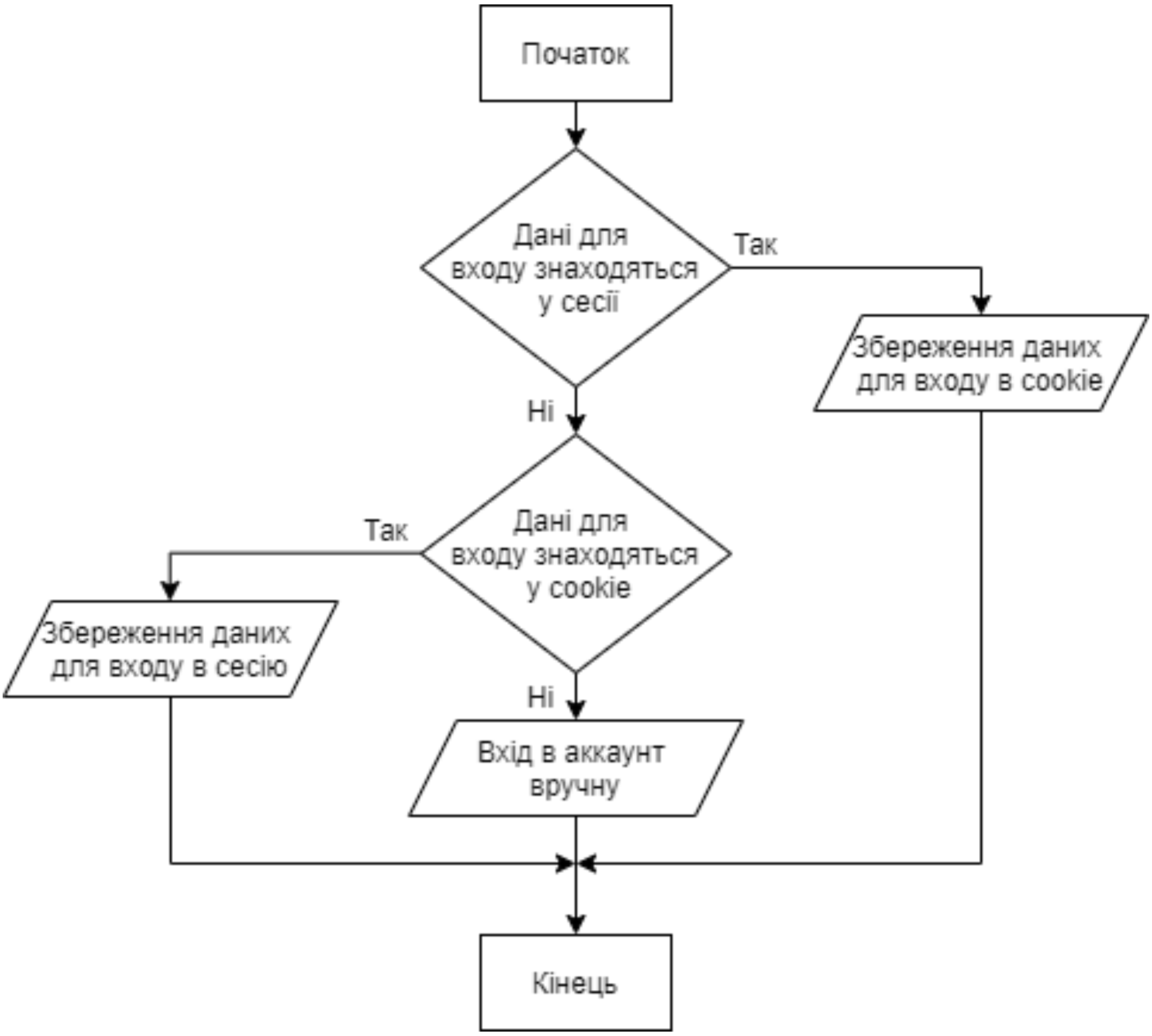

Рисунок 11 — Схема блоку роботи з cookie та сесіями.

Перш за все перевіряється чи є логін та пароль у сесії, якщо так, то сервіс зберігає дані з сесії у cookie та переправляє користувача на сайт, в іншому випадку додаток перевіряє наявність логіну та паролю у cookie, якщо там присутні шукані дані, то він зберігає їх у сесію та також перенаправляє користувача на сайт, а якщо ж і тут цих даних немає, то сервіс переводить користувача на сторінку логіну, де його просять повторно ввести дані для входу у сервіс.

Якщо поставити даний сервіс на хостинг, то ним можна буде одночасно користуватися з персонального комп'ютера, ноутбука, планшета та мобільного телефону одночасно, оскільки вся інформація синхронізується з хмарним сховищем.

2.7 Оцінка ефективності системи та варіанти її оптимізації

Для деяких великих систем може знадобитися проведення аналізу ефективності їх системи. Метою даного аналізу буде перерозподіл ресурсів, що виділяються у системі, наприклад, кількість терапевтів у лікарні в залежності від вимог, що представлються перед системою та вхідних даних таких, як інтенсивність надходження клієнтів, середній час обслуговування тощо.

Малим за розмірами, або одноразовим системам такий процес буде непотрібен, оскільки це зайва витрата матеріальних ресурсів, що не встигне окупитися за короткий проміжок роботи, або через малий потік клієнтів.

Великим системам же навпаки набагато вигідніше витратити додаткову кількість ресурсів для оптимізації процесу роботи після якого компанія почне набагато швидше отримувати прибуток.

Для прикладу розглянемо систему одноканального СМО, приймемо до уваги побажання клієнта та, якщо його система працює незадовільно дамо поради по оптимізації системи.

Клієнтом виступає лікарня, що спеціалізується на наданні терапевтичних послуг. Клієнт бажає щоб 85 відсотків людей, що звернулися до них за допомогою були обслужені.

Наразі у клієнта одночасно працює лише один лікар і він не може сам визначитись чи потрібно йому додатково наймати лікарів, чи ні. Для цього він звернувся до нас за допомогою.Заявки на вступ у чергу надходять з інтенсивністю λ, яка дорівнює 10 заявок на годину, а середня тривалість обслуговування лікаря $\overline{t_{об.}}$ = 7 хвилин.

Розглянемо приклад, який покаже наявно різницю багатоканального та одноканального СМО. Для простоти викладення скористаємося СМО з відмовами:

1) Визначити показники ефективності роботи системи масового обслуговування за наявності одного приймаючого лікаря;

2) Визначити оптимальну кількість приймаючих лікарів у лікарні, якщо умовою оптимальності вважати задоволення в середньому з кожних 100 заявок не менше 85 заявок будуть обслужені.

Розв'язання:

1) розрахуємо інтенсивність потоку обслуговування:

$$\mu = \frac{1}{\overline{t_{об.}}} = \frac{1}{7}\left(\frac{1}{\text{хв}}\right) = \frac{60}{7} = 8{,}57\left(\frac{1}{\text{год}}\right).$$

2) Визначимо відносну пропускну здатність СМО:

$$Q = \frac{\mu}{\lambda+\mu} = \frac{8{,}57}{10+8{,}57} = 0{,}462.$$

Це означає, що в середньому 46,2% заявок, які надходять, будуть задоволені й за ними будуть надані послуги, тобто їх огляне лікар.

3) Ймовірність відмови в обслуговуванні ($P_{\text{від.}}$) становитиме:

$$P_{\text{від.}} = \frac{\lambda}{\lambda+\mu} = \frac{10}{10+8{,}57} = 0{,}538.$$

Отже, в середньому 53,8% заявок, які надійдуть на обстеження, отримають відмову в обслуговуванні.

4) Абсолютна пропускна здатність СМО – лікарні дорівнюватиме

$$\text{A} = \frac{\lambda\mu}{\lambda+\mu} = \frac{10*8{,}57}{10+8{,}57} = 4{,}615.$$

Таким чином, в середньому за годину будуть обслужені 4,615 заявок на огляд.

З цього можна зробити висновок, що за наявності тільки одного приймаючого лікаря вимоги клієнта задоволено не буде, оскільки необхідною кількістю обслуговуваних заявок є 85% від надійшовшої кількості, звідси випливає той факт, що наявну кількість лікарів потрібно збільшувати, але як само клієнт теж не знає і знову просить нас допомогти. Давайте підрахуємо яку мінімальну кількість лікарів йому потрібно мати для задоволення своїх потреб.

Для виконання другого завдання задачі – визначення оптимального числа лікарів у поліклініці, слід перш за все проаналізувати інтенсивність навантаження каналу.

5) Обчислимо інтенсивність навантаження каналу:

$$p = \frac{\lambda}{\mu} = \frac{10}{8{,}57} = 1{,}167.$$

Тобто, за час середнього за тривалістю прийму 7 хвилин надходить в середньому 1,167 заявки на огляд.

6) Для одержання характеристик системи (поліклініки) та вибору оптимального варіанта кількості лікарів слід поступово збільшувати число каналів (лікарів) n = 2,3,4, перетворюючи таким чином наявну систему масового обслуговування з одноканальної в багатоканальну. Тоді відносна пропускна здатність становитиме:

$$Q = 1 - \frac{p^n}{n!} * p_0 = 1 - \frac{1{,}167^n}{2!} * 0{,}35 = 0{,}761;$$

$$p_0 = \left(1 + p + \frac{p^2}{2!} + \cdots + \frac{p^k}{k!} + \cdots + \frac{\rho^n}{n!}\right)^{-1};$$

При $n = 2;\ p_0 = \left(1 + 1{,}167 + \frac{1{,}167^2}{2!}\right)^{-1} = 0{,}35.$

Абсолютна пропускна здатність дорівнюватиме:

$$A = \lambda Q = 10 * 0.761 = 7{,}61$$

Аналогічно розрахуємо основні характеристики системи масового обслуговування для 3 та 4 каналів обслуговування (лікарів) та зведемо їх у табл. 5.

Таблиця 5 — Основні характеристики обслуговування заявок на огляд поліклінікою залежно від кількості лікарів

Характеристика	Кількість каналів (лікарів)			
	1	2	3	4
Відносна пропускна здатність (Q)	0,46	0,76	0,91	0,97
Абсолютна пропускна здатність (A)	4,62	7,61	9,15	9,76

Отже, це означає, що за годину будуть обслуговуватися в середньому 4,62; 7,61; 9,15; 9,76 заявки (A), тобто 46, 76, 91 та 97 (%) звернувшихся за допомогою її отримають, при $n = \overline{1..4}$ відповідно.

Після проведеного аналізу можемо зробити висновок, що для задоволення щонайменше 85% надійшовших до клієнта заявок з поданими початковими умовами йому необхідно мати трьох та більше лікарів одночасно.

Висновки до розділу 2

Даний сервіс розроблено притримуючись RESTful правил, що надало можливість легко масштабувати розроблену систему, однозначно та, що є вкрай важливо, зрозуміло описувати систему та усі можливості переходу користувача по частинам сервісу, працювати зі зручними типами даних не витрачаючи зайвий час на обробку вхідних даних.

Завдяку даному підходу дуже зручним є описання API запитів, що дозволяє працювати із сервісом із будь-яким графічним інтерфейсом, що забажає бачити замовник, або навіть взагалі без графічного інтерфейсу, якщо мова буде йти про підключення модерування системи за допомогою консолі або терміналу.

Усі записи даного API покриваються за допомогою чотирьох основних типів запиту таких, як GET, POST, PUT та DELETE, але в силу абстрактності даних методів усі запити можна покрити двома типами: GET та POST.

Даний підхід буде зрозумілий людям без технічної освіти, що є вкрай важливим параметром для будь-якого клієнта, аби він не відчував дискомфорту при роботі із сервісом.

Іншою, вкрай важливою, технологією, що дозволила створити легковажний, масштабований та універсальний сервіс є мікрофреймворк Flask, що є імпортованою бібліотекою у мові програмування Python.

Головна перевага даного портативного елементу — це можливість використовувати всі його переваги та функції не віднімаючи можливість використовувати інші бібліотеки для описання інших функцій.

В свою чергу всі компоненти були підібрані так, щоб вони не конкурували один із одним та виконували свою основні функції якомога краще.

Так наприклад для роботи з БД обрано окремий інструмент, що під'єднується до БД Oracle 11g та забезпечує стабільну роботу із усіма сутностями системи.

Завдяки даному фреймворку була налаштована додаткова перевірка введеної інформації на стороні клієнта, аби зайвий раз не навантажувати систему захищаючись від простих спроб атаки на розроблений сервіс.

Але перевірка вхідних даних на стороні клієнта є необхідною, але не достатньою умовою верифікації даних, що надходять до системи, оскільки хакери можуть за допомогою спеціальних команд виконаних у браузері призупинити виконання необхідних перевірок на стороні клієнта, тому у розробленому сервісі також присутня перевірка на стороні серверу, куди зловмисник не буде мати доступу, що в свою чергу не надасть йому можливості атакувати систему викрадаючи, змінюючи, або видаляючи дані, що знаходяться у системі.

У якості типу БД обрано SQL систему, оскільки її недоліки є несуттєвими у порівнянні із недоліками NoSQL.

Основним аргументом є дублювання даних у системі, при спробі створення нових таблиць, оскільки дані не мають чіткої схеми.

Насамперед, у даному випадку, при описанні сервісу електронної черги головний недолік SQL типу не є недоліком, оскільки збереження такої кількості продубльованих даних є недоцільним.

Дана БД була спроектована з урахуванням стандартних нормальних форм, що дозволяє користуватися розробленою системою для зберігання даних впродовж тривалого часу без втрати інформації з БД та легкої можливості розширення БД.

Розширювати SQL БД краще за все за допомогою додавання суміжних таблиць, що містять лише зовнішні ключі інших необхідних додаткових таблиць, що раніше не були в системі.

Приклад додавання таких суміжних таблиць детально показано у підрозділі 2.4, а саме таблиці 3, де показано як з двох атомарних таблиць «Користувачі» та «Продукти» формується допоміжна таблиця «Замовлення».

Поля «Номер» у таблицях «Користувачі» та «Продукти» є головним ключем, що переходять у таблицю «Замовлення» становлячись зовнішніми ключами.

Завдяки даному підходу при побудові таблиць поля «Електронна пошта» та «Ім'я» з таблиці «Користувачі» та поля «Назва» та «Ціна» з таблиці «Продукти» не дублюються у таблиці «Замовлення».

В той же час завдяки транзитивним зв'язкам ці дані не втрачаються.

Також у даному розділі представлено представлено схему роботи клієнта із сервісом, що більш детально показано на рисунках 6 — 11.

Однією із переваг є числова візуалізація ефективності системи та способи її оптимізацію, що показані на прикладі роботи лікарні.

Завдяки даному функціоналу великі системи зможуть ефективніше надавати послуги користувачам не витрачаючи зайві ресурси у напрямах, де вони непотрібні.

РОЗДІЛ 3. ОСОБЛИВОСТІ АРХІТЕКТУРНО-СТРУКТУРНОЇ ОРГАНІЗАЦІЇ СЕРВІСУ РОЗРОБЛЕНОГО СЕРВІСУ ЕЛЕКТРОННОЇ ЧЕРГИ

3.1 Опис роботи компонентів розробленого сервісу

Як вже було описано в попередніх розділах, даний сервіс написано на мові Python, що надало можливість використовувати величезну кількість різноманітних відкритих бібліотек для виконання тих чи інших задач, таких як реалізація програмної частини серверу, відтворення графічного інтерфейсу для зручного користування сервісом, валідації даних як на стороні клієнту, так і бібліотеки, що забезпечує підключення сервісу до бази даних Oracle 11g, де й зберігається уся необхідна для роботи сервісу інформація.

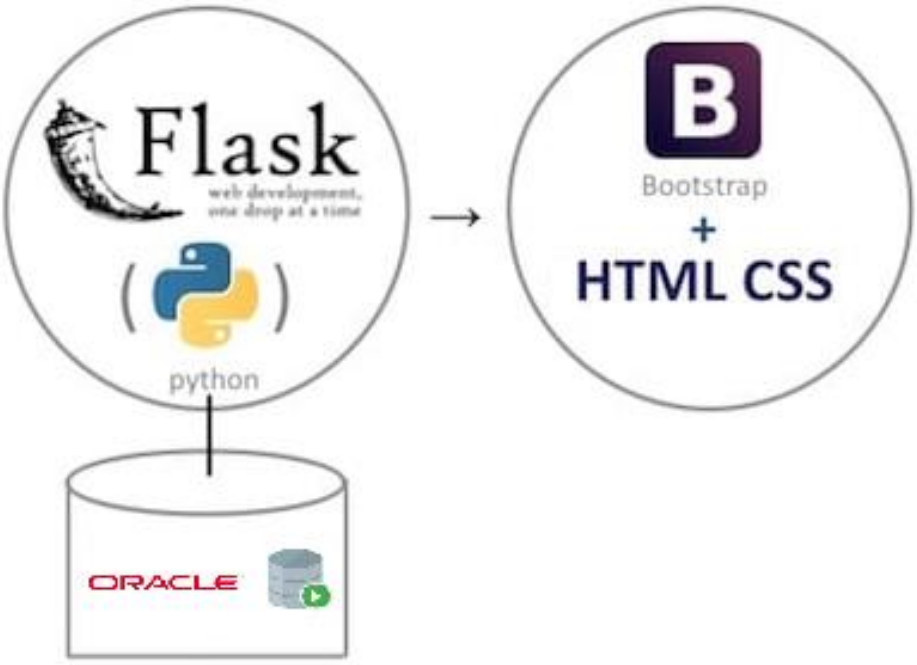

Рисунок 12 — Схема роботи сервісу

Коли користувач заходить на домашню сторінку сайту він відправляє GET запит на API, Python код перехоплює даний запит та запускає відповідний метод,

що помічений анотацією @app.route('/') у якому описаний процес, що має відбутися.

В ниншій реалізації відбувається рендер домашньої сторінки. Увесь функціонал з'являється на сторінці за допомогою WTForms, а уся інформація, що була виведена структурується та виводиться у зрозумілому, інтуітивнозручному та красивому інтерфейсу стає саме такою завдяки бібліотеці Bootstrap.

Якщо ж користувач зайде на сторінку, де будуть знаходитися якісь поля для заповнення, то при першому переході на цю сторінку він відправить GET запит на відображення відповідної сторінки.

Якщо ж користувач спробує оновити таку сторінку після того, як він на неї потрапить, то він відправить POST запит на зміну даних у БД. Але завдяки валідації зі сторони клієнта, що була додана завдяки використанню бібліотеки WTForms такий запит не піде у БД через програмну частину сервісу.

Якщо користувач спробує ввести невалідні дані й натисне на кнопку, щоб перейти далі, то в цей момент на сервер відправиться POST запит, але перед тим як продовжити виконання коду відбудеться перевірка полів, на які була накладена валідація, яка покаже яке саме поле не пройшло перевірку. Поле стане червоним та на ньому відобразиться спеціальне повідомлення, у якому буде чітко вказано вимоги до його заповнення.

Якщо користувач введе дані коректно, то після того як валідація на клієнті буде успішно пройдена WTForms предасть дані до БД, для подальшої валідації на сервері. У разі успішною валідації на сервері дані будуть оброблені та відповідним чином передані до БД для подальшої роботи.

Можуть статися випадки, коли потрібно буде працювати з БД без участі сервісу. Наприклад, обробити велику кількість даних, що є не дуже зручним при роботі через графічний інтерфейс, оскільки даний сервіс був розроблений зовсім для інших потреб.

Для вирішення задачі такого типу можна використати майже будь яку інтегровану середу розробки, що надає змогу працювати із БД. За рахунок того, що у якості БД обрано рішення від Oracle, нам надається можливість використовувати різні середовища на свій розсуд.

Завдяки цьому розробник зможе легко оперувати великими об'ємами даних не використовуючи для цього розроблений сервіс.

3.2 Виконувані функції

За допомогою даного сервісу можна: зареєструватися, увійти в аккаунт, вийти з аккаунту, подивитися інформацію про користувачів, змінити інформацію про користувачів, видалити користувачів, створити статуси, подивитися статуси, змінити інформацію про статуси, видалити статуси, створити місця, подивитися місця, змінити інформацію про місця, видалити місця, створити події, подивитися події, змінити інформацію про події, видалити події, створити події для черг, подивитися події для черг, змінити інформацію про події для черг, видалити події для черг, створити черги, подивитися черги, змінити інформацію про черги та видалити черги.

Висновки до розділу 3

Провівши аналіз даних у попередніх підрозділах та обґрунтувавши вибір стеку технологій прийнято рішення писати обидві клієнтську та серверну частини сервісу переважно за допомоги мови програмування Python та суміжних з нею бібліотек таких, як Flask, WTForms, Bootstrap та Validators які чудово виконують поствлені їм задачі не заважаючи роботі один одного.

У якості архітектурного принципу обрано REST за рахунок його легковажності, масштабованості та простоти.

Сховищем даних обрано SQL БД Oracle 11g, що дозволяє економити місце на сервері, внаслідок чого пришвидшити роботу сервісу, оскільки запити у БД та з неї потребуватимуть менших обчислювальних витрат.

Так як даний сервіс будувався з урахуванням можливих архітектурних змін, його зміна чи оновлення буде доволі легким за рахунок того, що сервіс виступає у роли купи мікросервісів, що об'єднані в одну систему, а не один великий моноліт.

Це дає можливість легко додати нову бібліотеку, замінити її, або навпаки видалити, якщо у ній більше немає потреби.

Також продумано ситуацію, при якій адміністратору БД знадобиться працювати з великими об'ємами даних. Для цього передбачено роботу з БД за допомогою інтегрованої середи розробки, що була додана для роботи з різними БД.

Так, як у якості БД обрано Oracle 11g, то така БД підтримується майже у всіх інтегрованих середах розробки.

Для опрацювання великих об’ємів даних для даного сервісу будуть використовуватися такі SQL запити, що надають можливість працювати з

окремими таблицями, або з деякими вибірками цих таблиць за допомогою таких команд, як: SHOW, CREATE, USE, SOURCE, INSERT, UPDATE, DELETE, DROP, SELECT, JOIN та інших.

Описані синтаксисом Oracle 11g SQL команди дозволяють виконати будь-яку дію з даними, що знаходяться у БД.

РОЗДІЛ 4. ОПИС ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ РОБОТИ СЕРВІСУ ЕЛЕКТРОННОЇ ЧЕРГИ

4.1 Тестування розробленого програмного забезпечення

Як було описано вище валідація відбувається як на клієнті (за допомогою HTML), так і на сервері (за допомогою WTForms).

На рис. 13, рис. 14 та рис. 15 показано приклад валідації на клієнті за допомогою HTML, а саме атрибуту "type" тегу "input". Щоб перевірити виконання перевірки введених даних потрібно ввести некоректну інформацію та зробити спробу пройти далі.

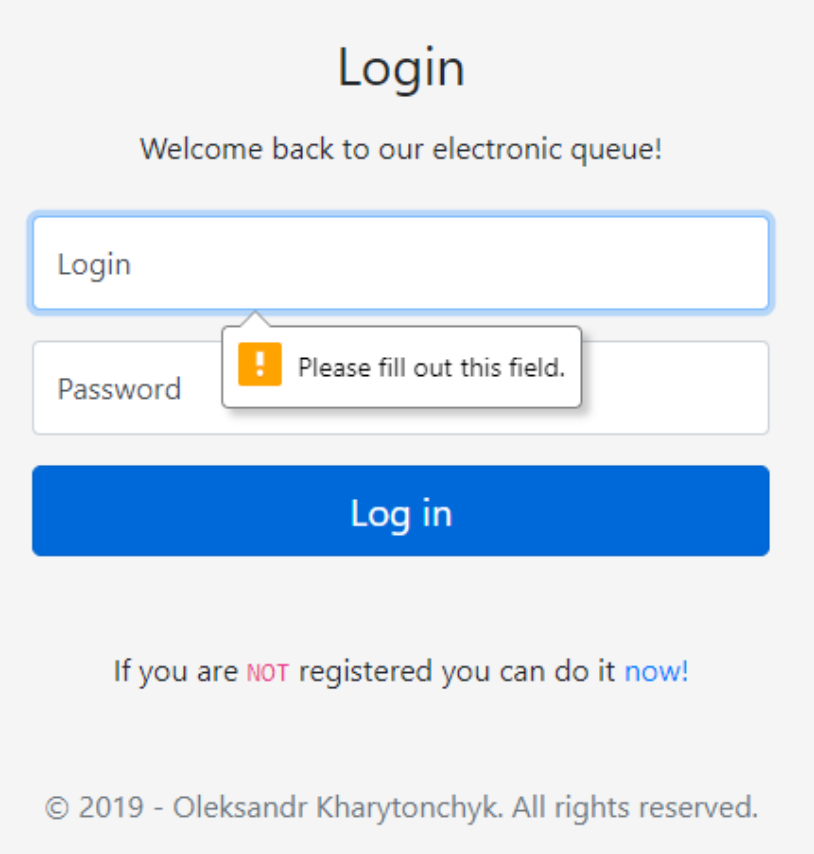

Рисунок 13 — Перевірка інформації введеної користувачем на стороні клієнта на сторінці авторизації.

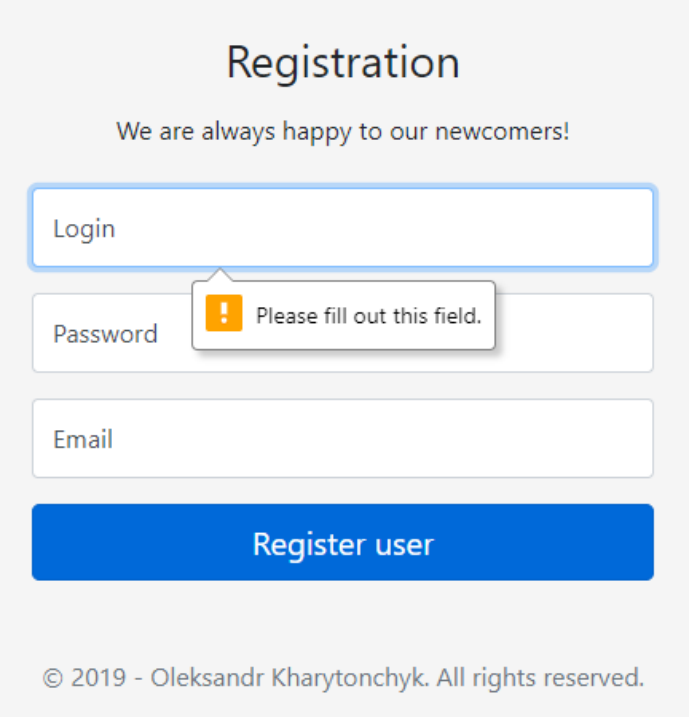

Рисунок 14 — Перевірка інформації введеної користувачем на стороні клієнта на сторінці реєстрації при спробі продовження без введених даних.

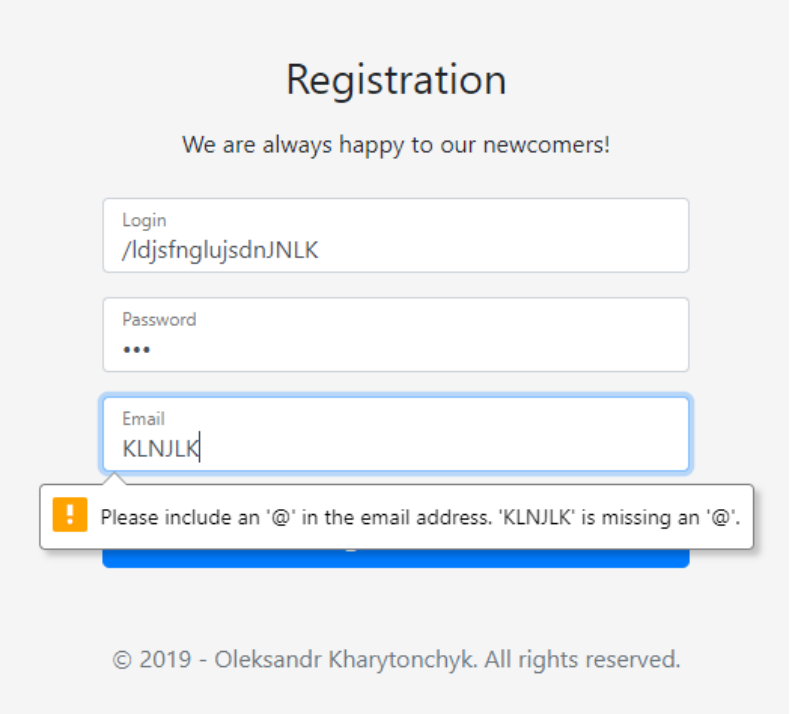

Рисунок 15 — Перевірка інформації введеної користувачем на стороні клієнта на сторінці реєстрації при спробі продовження з некоректними даними.

На рис. 16, рис. 17 та рис. 18 показано приклад валідації на сервері за допомогою WTForms, а саме атрибуту "Data.Required" тегу "input". Щоб перевірити виконання перевірки введених даних потрібно ввести некоректну інформацію та зробити спробу пройти далі.

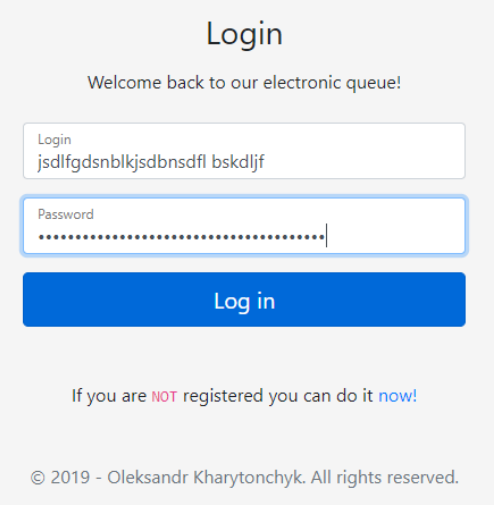

Рисунок 16 — Вікно для вводу інформації користувачем на сторінці авторизації.

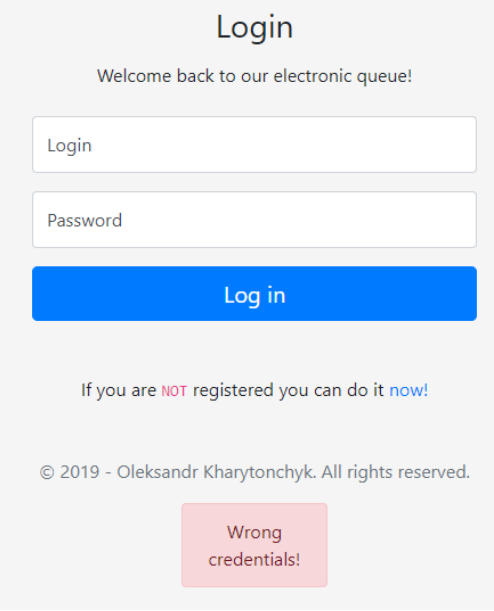

Рисунок 17 — Перевірка інформації введеної користувачем на стороні клієнта на сторінці авторизації при спробі входу під не зареєстрованим користувачем.

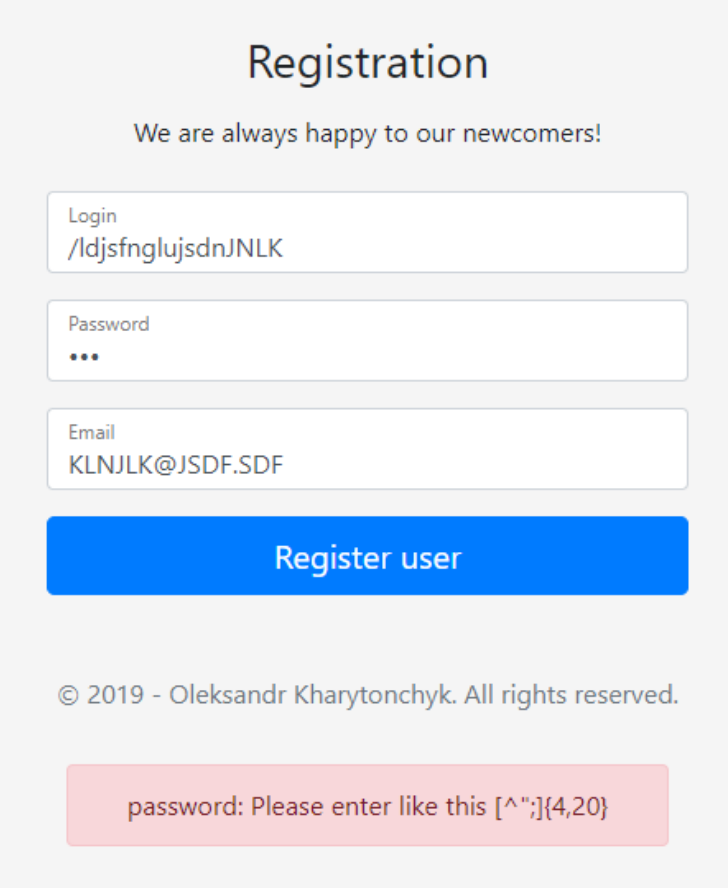

Рисунок 18 — Перевірка інформації введеної користувачем на стороні клієнта на сторінці реєстрації при спробі зареєструвати користувача із некоректними даними.

4.2 Огляд функціональних можливостей розробленої системи

Розроблений сервіс дозволяє виконувати всі основні дії з описаними його компонентами, такі як: створення, отримання, оновлення та видалення таких сутностей, як: користувачі, статуси, місця, події, створені події у місці та черги.

Покажемо основний спосіб використання розробленого сервісу. Для цього користувачу потрібно зайти на сайт, зареєструватися, та стати у чергу до лікаря:

По-перше: користувачу потрібно зайти на сайт.

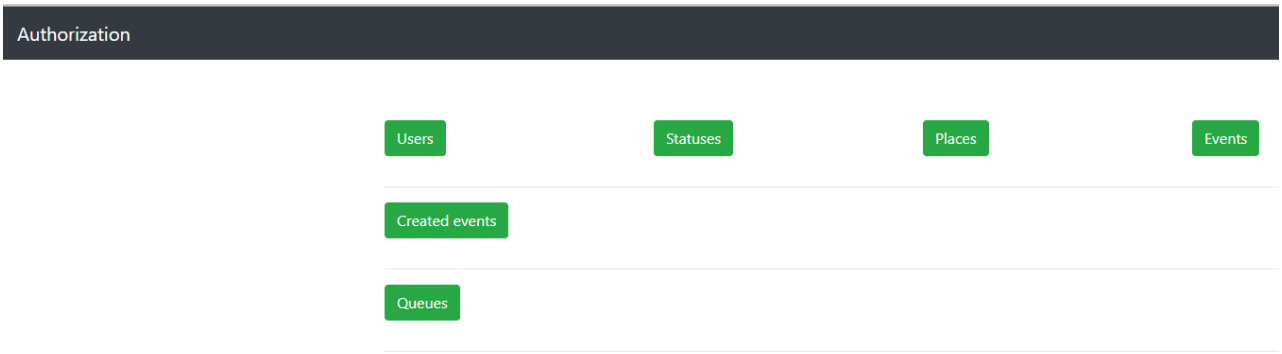

Рисунок 19 — Головна сторінка сайту.

По-друге: йому потрібно перейти на сторінку авторизації.

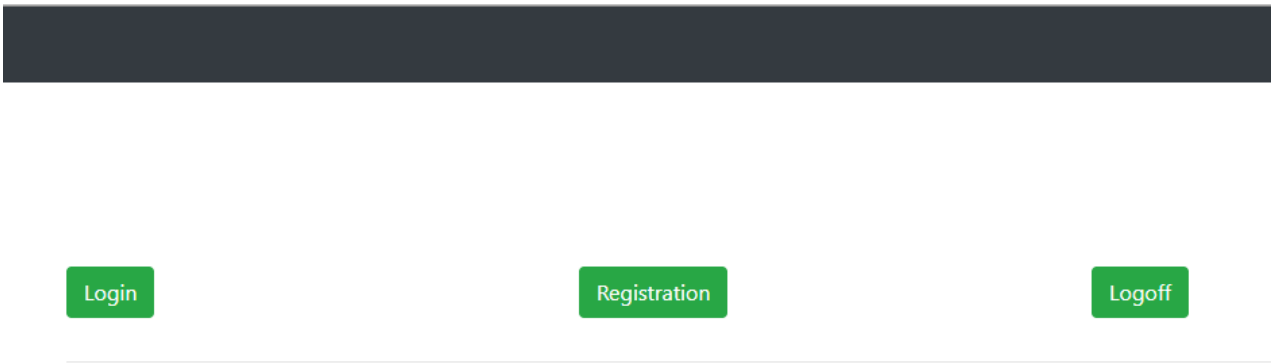

Рисунок 20 — Сторінка авторизації сайту.

По-третє: йому потрібно зареєструватися на сайті.

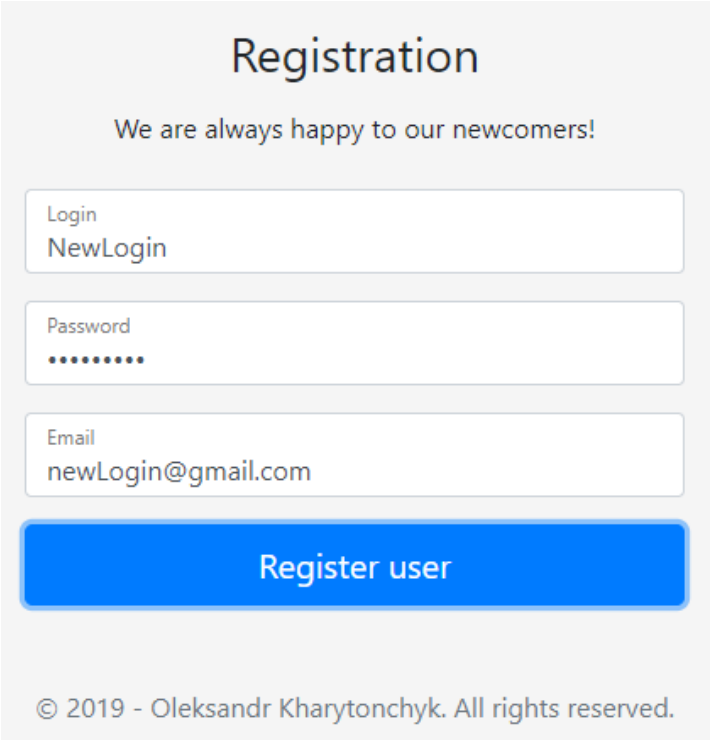

Рисунок 21 — Сторінка реєстрації на сайті.

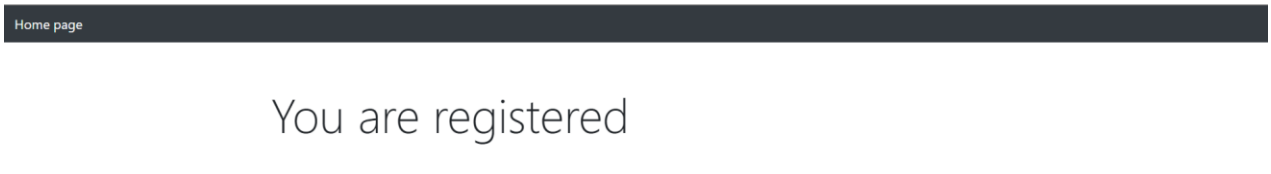

Рисунок 22 — Інформаційне повідомлення після підтвердження реєстрації.

По-четверте: користувач може перевірити можливість ставання у чергу до лікаря.

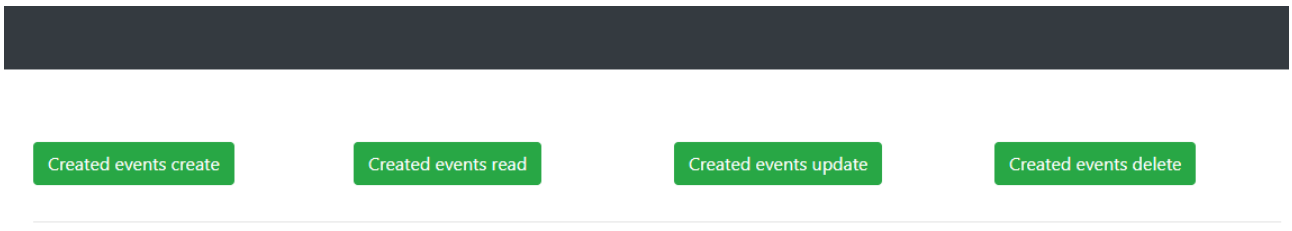

Рисунок 23 — Сторінка для роботи із подіями на які можна стати у чергу.

Для цього він може вивести список подій.

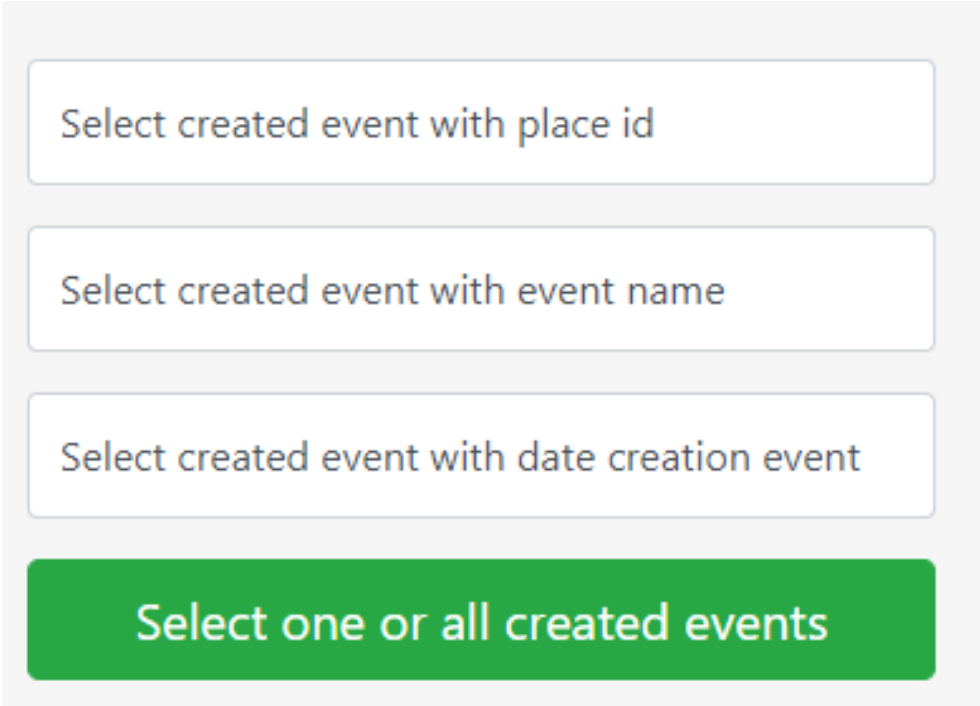

Рисунок 24 — Сторінка для перегляду інформації про події на які можна стати у чергу.

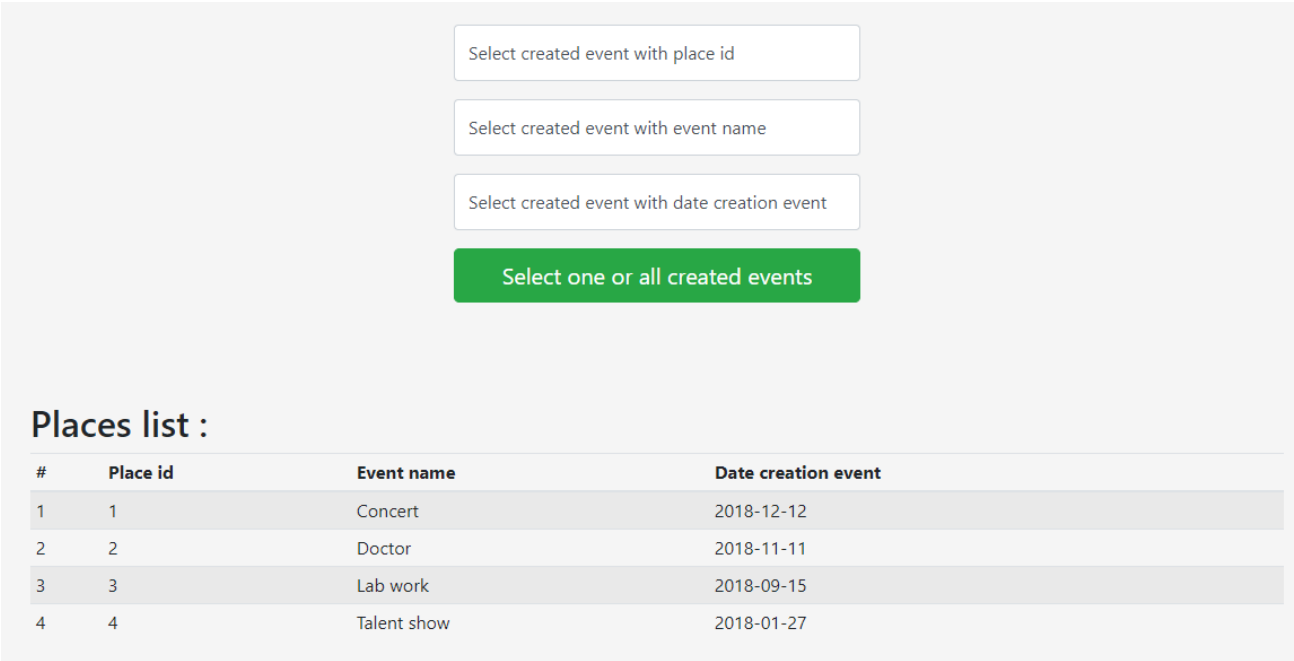

Рисунок 25 — Список подій на які можна стати у чергу.

По-п'яте: користувач додасть себе у чергу до лікаря, вказавши необхідні дані.

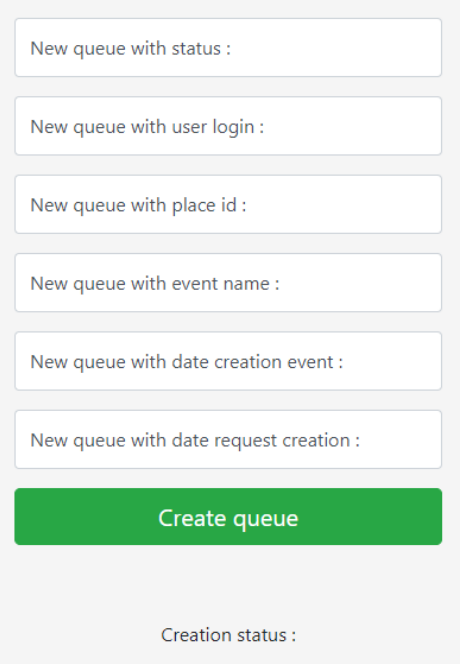

Рисунок 26 — Сторінка для додавання у чергу.

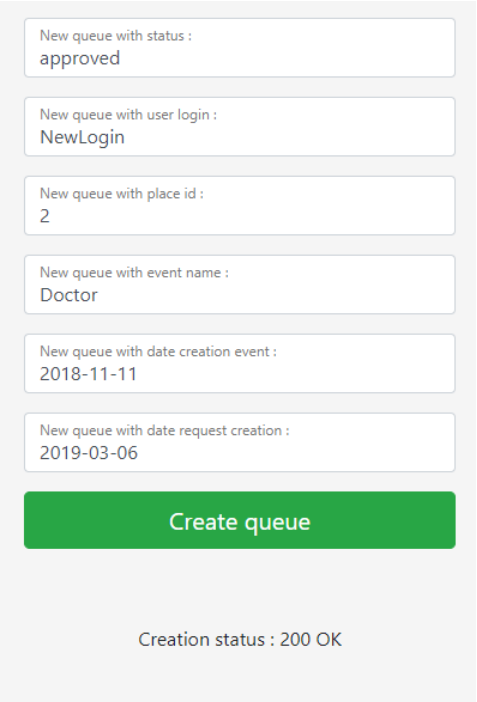

Рисунок 27 — Сторінка для додавання у чергу із введеними даними.

По-шосте: користувач може перевірити, що він дійсно став у чергу.

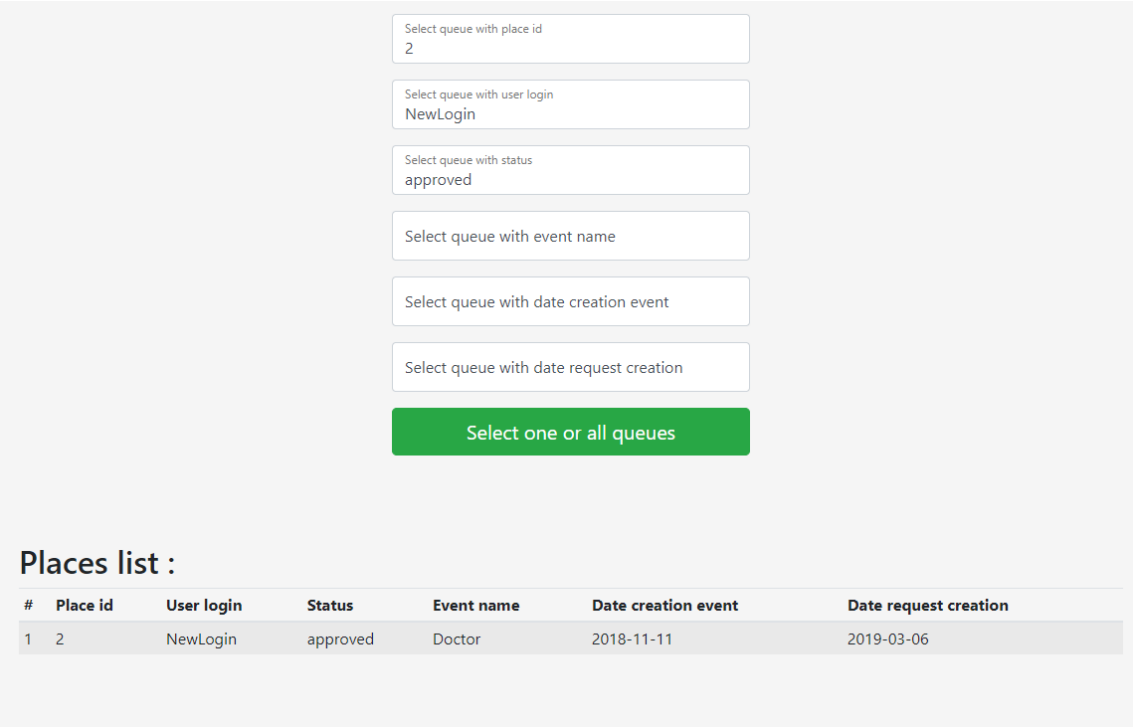

Рисунок 28 — Сторінка для перегляду черг із введеними даними.

Тепер він може відредагувати свою заявку у черзі змінивши дату прийому:

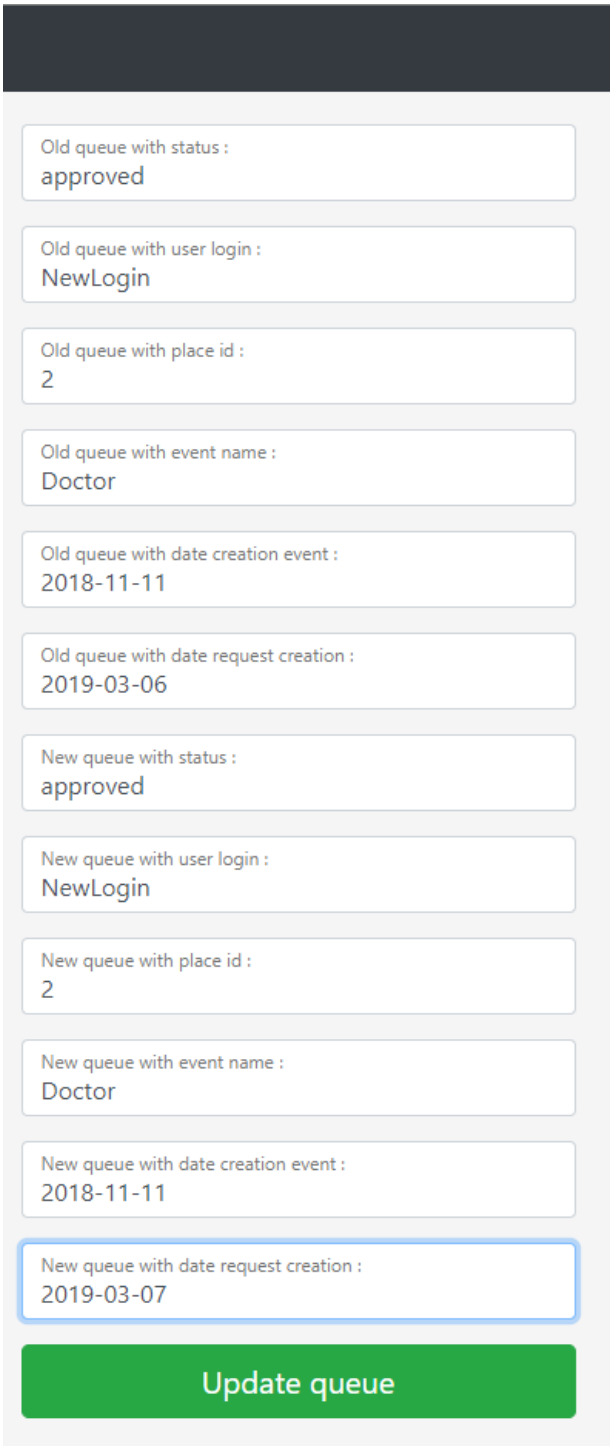

Рисунок 29 — Сторінка для редагування черги.

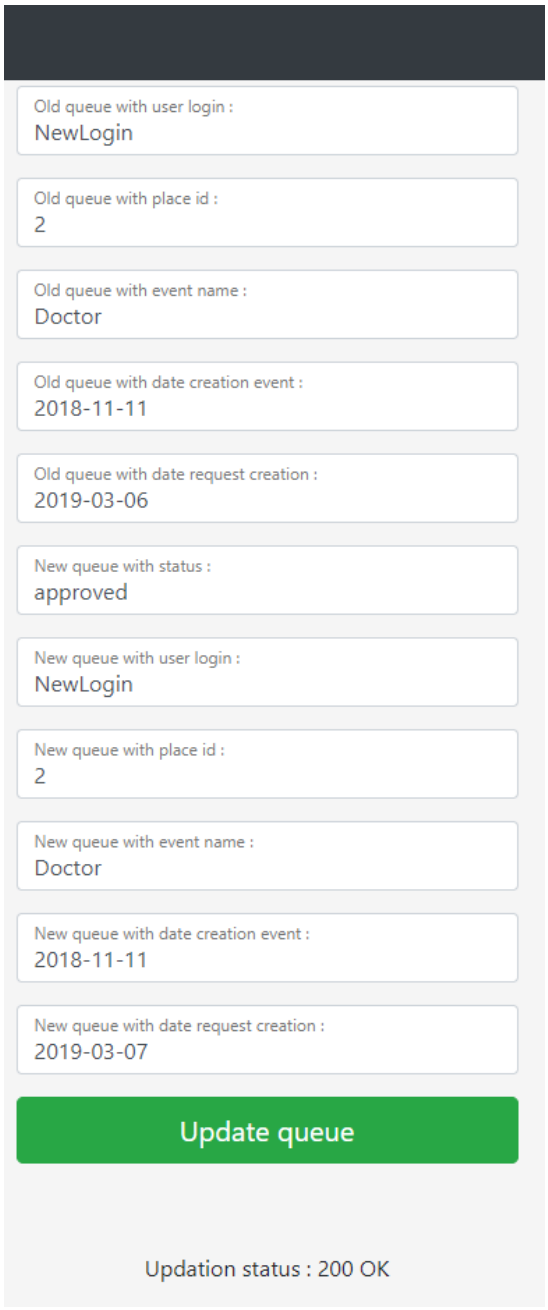

Рисунок 30 — Сторінка для редагування черги.

Також коирстувач може перевірити, що він дісно відредагував запис у черзі.

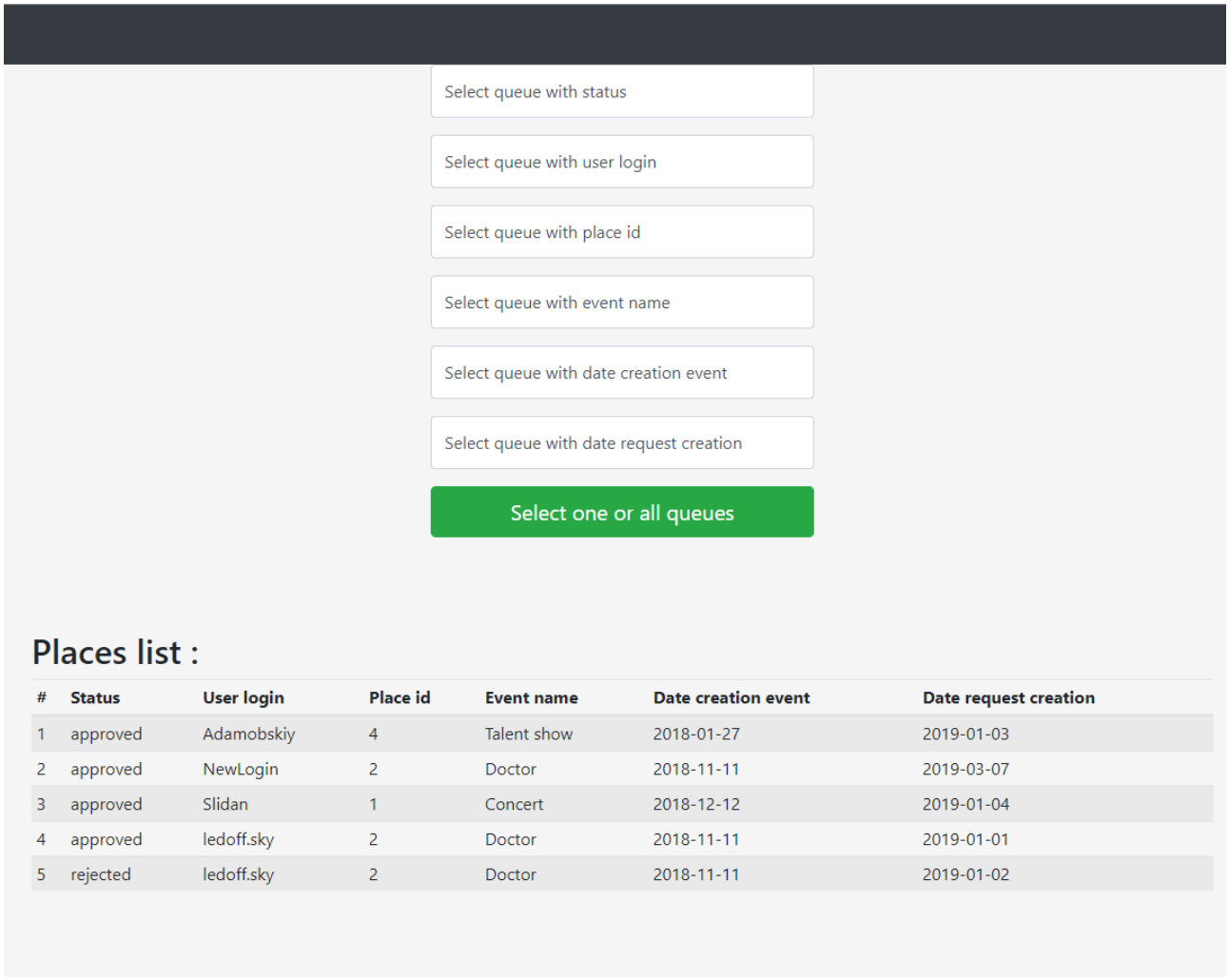

Рисунок 31 — Сторінка для перегляду черг.

Якщо йому знадобиться видалити свій запис із списку черг до лікаря, то він зможе це зробити відповідним чином.

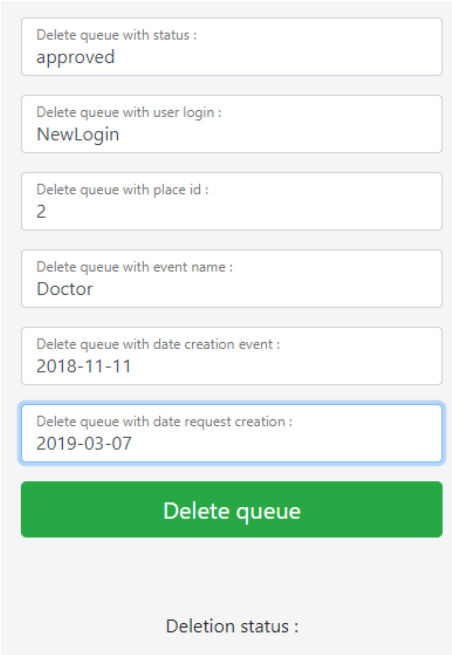

Рисунок 32 — Сторінка для видалення черг.

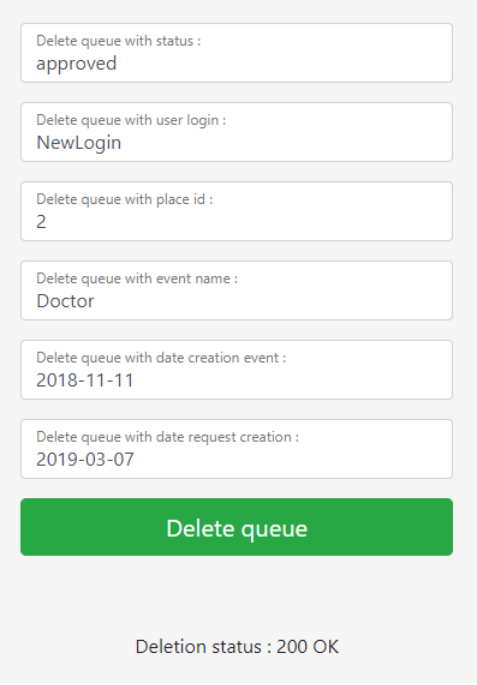

Рисунок 33 — Сторінка для видалення черг із введеними даними.

Користувач може перевірити, що він дійсно видалив запис із списку черг.

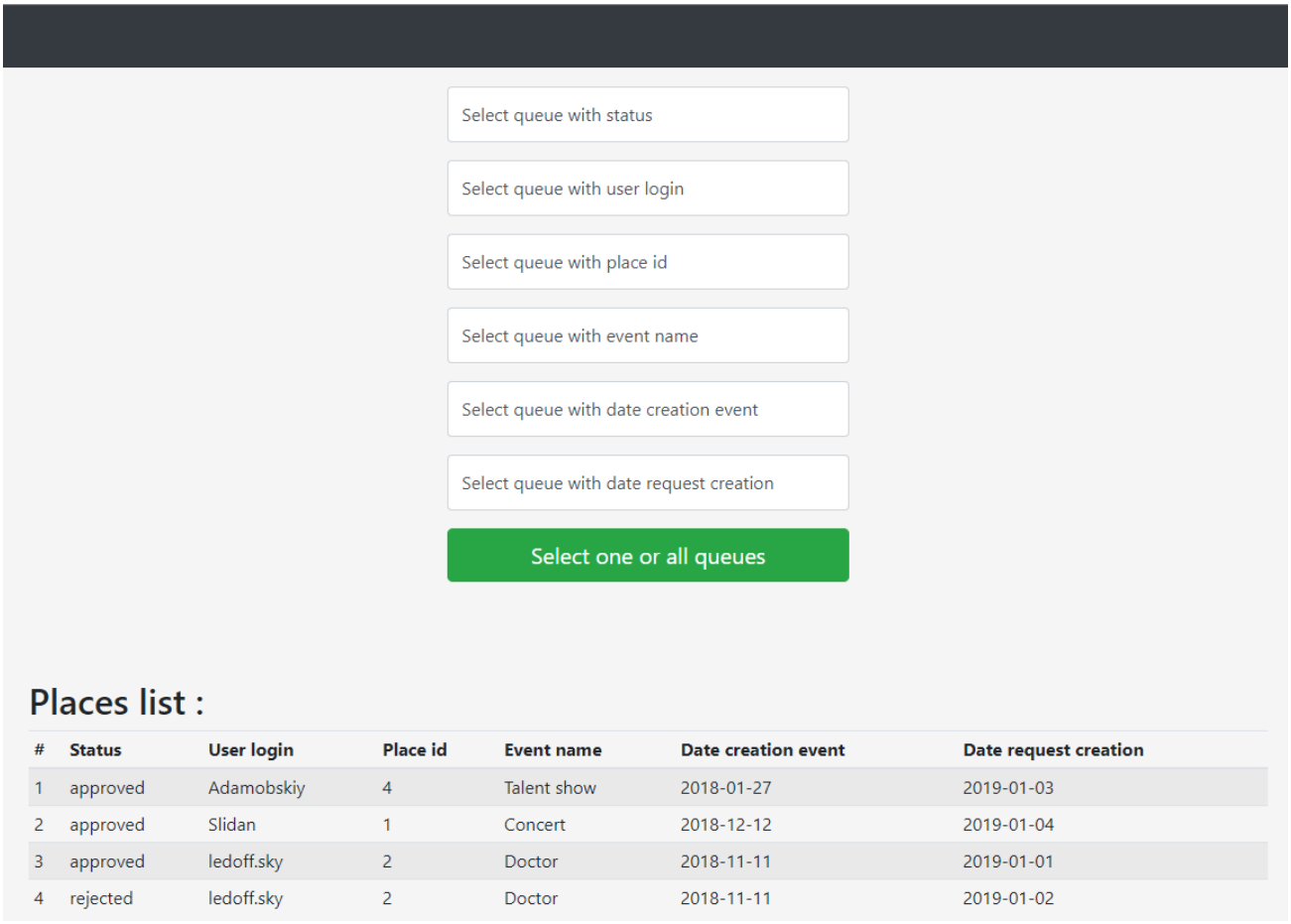

Рисунок 34 — Сторінка для перегляду черг.

На даному прикладі вдалося показати повний цикл роботи із записами у черзі, що і є основним завданням сервісу. Якщо бажаний користувачем час буде зайнято, то він не зможе стати на нього у чергу, що є звичайною практикою.

Також по бажанню замовника можна додати функціонал СМС-сповіщення про статус у черзі.

Для забезпечення зручності користування сервісом була додана додаткова можливість роботи з cookie та сесіями браузеру для того, щоб користувач не повинен був постійно входити у свій аккаунт при кожному перезапуску браузера.

При переході на сайт вперше у браузері сворюється сесія, яка зберігає в собі дані про користувача.

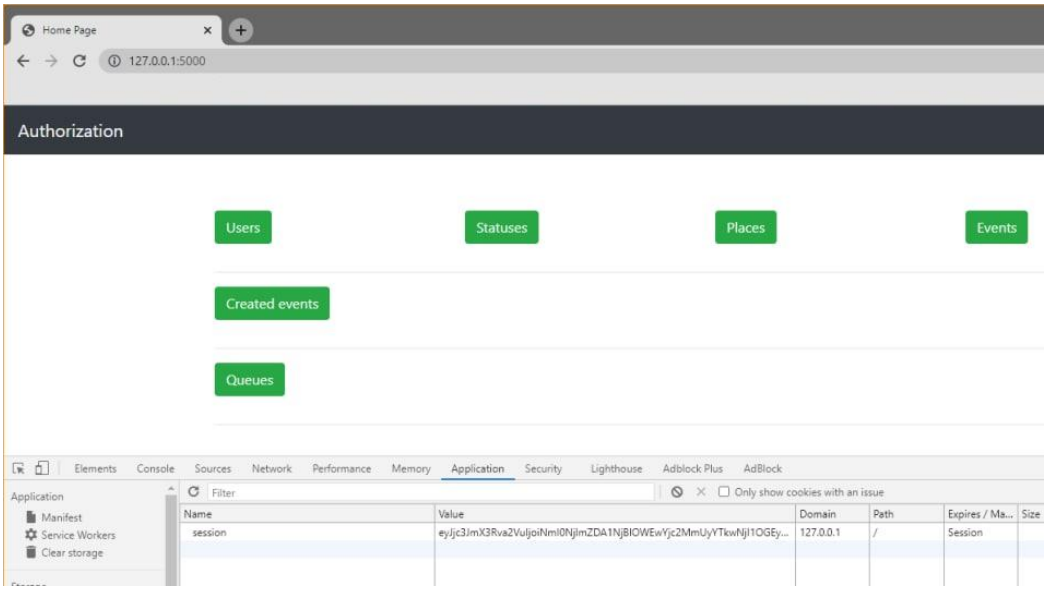

Рисунок 35 — Перегляд даних, що зберігаються у браузері під час першого входу на сайт.

Якщо користувач перейде на сторінку логіну з «чистою» сесією, то йому відобразиться сторінка входу у застосунок.

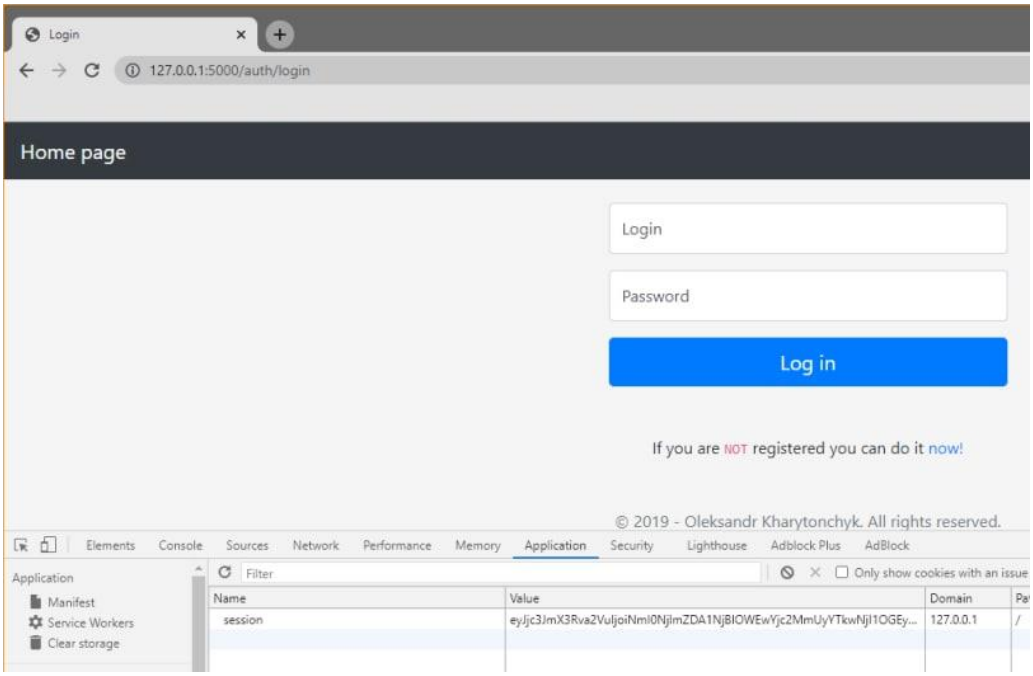

Рисунок 36 — Відображення інтерфейсу користувача при переході на сторінку логіну з «чистою» сесією.

Після реєстрації у браузері збережется cookie та сесія з даними для верифікації користувача.

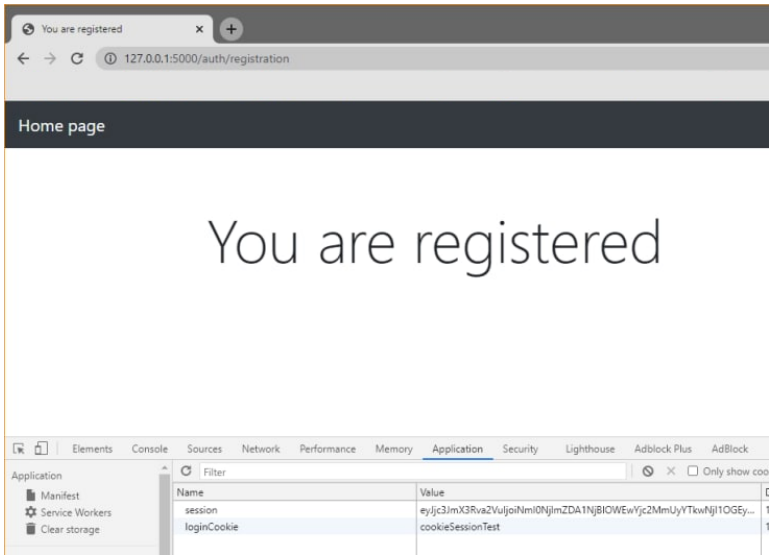

Рисунок 37 — Збережені дані у браузері після реєстрації.

Якщо користувач перейде на сторінку логіну з сесією, яка матиме дані для верифікації користувача, то йому відобразиться повідомлення про те, що він вже увійшов у свій аккаунт за допомогою сесії.

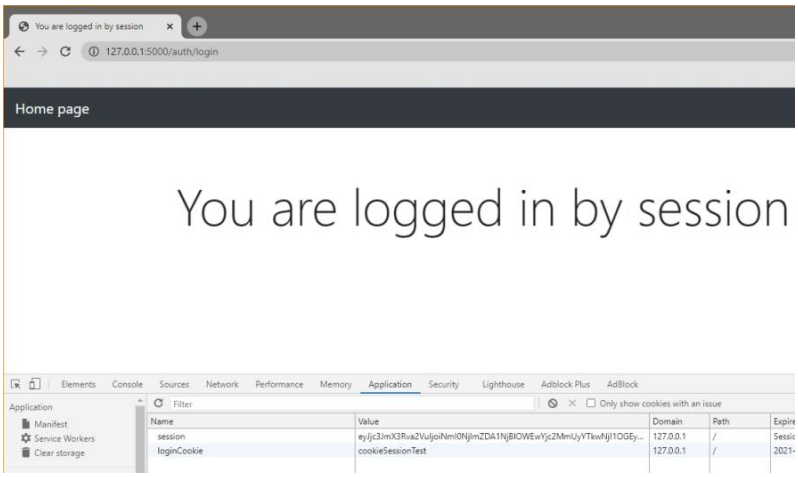

Рисунок 38 — Відображення інтерфейсу користувача при переході на сторінку логіну з сесією, що має дані для верифікації користувача.

Може статися ситуація, при якій сесія користувача буде втрачена. Її можна зімітувати за допомогою видалення сесії вручну.

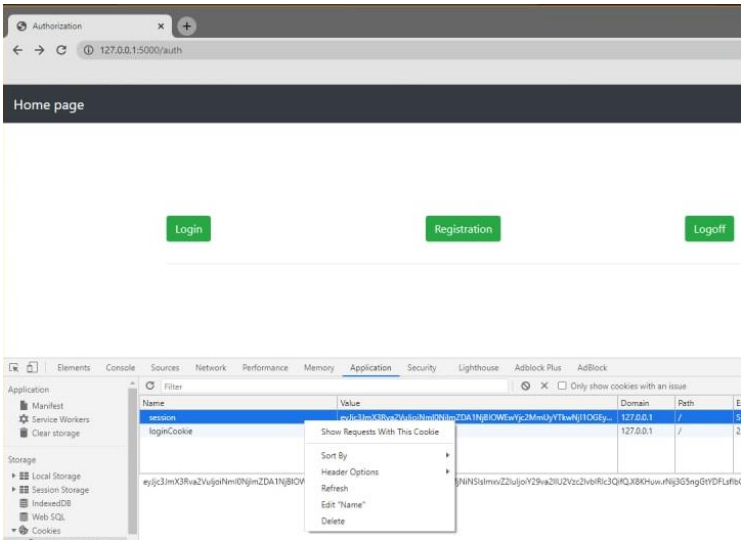

Рисунок 39 — Видалення сесії вручну.

Якщо користувач перейде на сторінку логіну без сесії взагалі, але у його браузері буде cookie для верифікації користувача, то йому відобразиться повідомлення про те, що він вже увійшов у свій аккаунт за допомогою cookie та в сесію браузера збережуться відповідні дані.

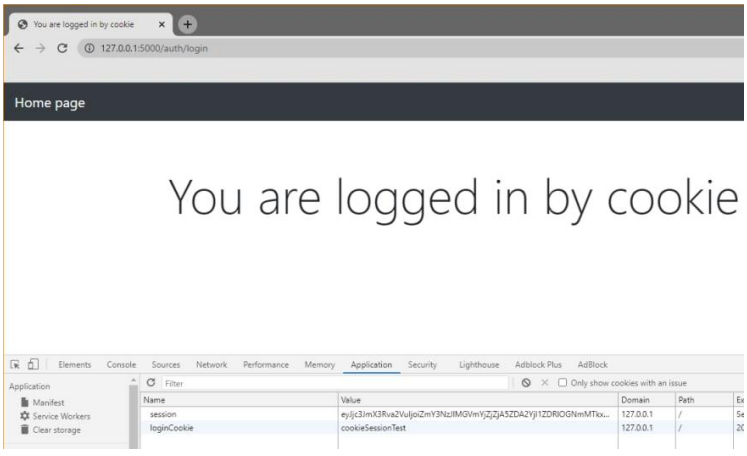

Рисунок 40 — Відображення інтерфейсу користувача при переході на сторінку логіну з cookie, що має дані для верифікації користувача без сесії.

Якщо користувач спробує перейти на сторінку логіну з сесією, що містить дані для верифікації користувача, але без cookie, то вони автоматично додадуться до браузера після обробки.

Якщо користувач перейде до сторінки виходу із аккаунту, то із його сесії буде видалено дані для верифікації юзера як уже зареєстрованого користувача а, cookie буде видалено повністю, оскільки вони мають тільки дані для верифікації користувача.

Висновки до розділу 4

У даному розділі показано приклад роботи із сервісом, що є повністю протестованим для надійного користування.

Розроблений сервіс є універсальним не тільки за рахунок того, що він надає можливість ставати у будь-які черги, а й за рахунок того, що його можна використовувати за допомогою будь-якого пристрою, що має доступ до інтернету завдяки браузеру, що може бути встановлений майже на будь-який пристрій від холодильника до персонального комп'ютера.

Даний функціонал вдалося реалізувати завдяки адаптивному дизайну, що автоматично змінюється відповідно до розміру вікна браузера. Це реалізовано шляхом зменшення, або збільшення розмірів відповідних кнопок на сайті у процентному співвідношенні до розмірів вікна браузера.

Даний застосунок є доступним з будь-якого браузера, оскільки технології, що використовувались при написанні графічного інтерфейсу користувача були обрані універсальні, під останні версії браузерів на теперішній момент.

Також показано роботу сервісу із cookie та сесіями, що забезпечує зручне користування сервісом за рахунок того, що користувач не повинен постійно входити у свій аккаунт, якщо він з нього не виходив.

Якщо користувач закриє браузер не виходячи попередньо зі свого аккаунту, то при наступному вході на сайт дані у сесію браузера перенесуться з cookie, що за умовами за замовчуванням зберігається 90 днів. Цей параметр можна змінити за бажанням користувача.

ВИСНОВКИ

У ході вирішення поставленої задачі магістерської дисертації розроблено нову систему, що збільшує ефективність програмно-апаратних засобів сервісу електронної черги за рахунок зменшення часу очікування у черзі та реалізації нових функціональних можливостей дистанційної взаємодії із засобами цього сервісу, іншими словами зменшено час очікування у черзі та надано можливість записатися у чергу дистанційно, тим самим покращена якість обслуговування клієнтів, шляхом зменшення часових витрат на впорядкування потоку нових користувачів, бажаючих скористуватися деякими послугами за допомогою розробленого математичного та ПЗ відповідного сервісу.

Розроблено новий метод, що надає можливість стати у чергу у будь-яке місце, а не тільки у те, де запроваджено власну, замкнену та не надійну систему.

Це досягнуто завдяки можливості легкого додавання будь-якого місця на яке користувач бажає стати у чергу та надання можливості це зробити з будь-якого пристрою, що може під’єднатися до мережі інтернет за допомогою браузера. Такими пристроями можуть слугувати: холодильники, годинники, ваги, тостери, принтери, кондиціонери, телефони, планшети, ноутбуки, персональні комп’ютери та інші.

Усі ці можливості отримано за рахунок продуманого вибору інструментів поміж усіх інших, за допомогою яких створено відповідний сервіс. Цими технологіями є мова програмування Python, мікрофреймворк Flask, бібліотеки WTForms, Bootstrap та Validators, архітектуру REST, а у якості БД обрано реляційну Oracle 11g.

БД була спроектована з урахуванням стандартних нормальних форм, що дозволяє користуватися розробленою системою для зберігання даних впродовж тривалого часу без втрати інформації з неї та легкої можливості розширення БД.

З нею зручно працювати як за допомогою розробленого сервісу, так і за допомогою інтегрованої середи розробки, що створена для роботи з різними БД.

Якщо користувач бажає покращити свій бізнес за допомогою підвищення ефективності шлях перерозподілу ресурсів, то він може це отримати за рахунок СМО, що у цифрах покаже як вигідніше зробити відповідний перерозподіл.

В якості математичної моделі обрано само СМО для роботи з чергою, оскільки поміж інших вона краще підходить для поставленої задачі за рахунок її направленості.

Даний сервіс дозволяє переглядати усіх користувачів у черзі, форматувати чергу в будь-які місця обслуговування, які створено у сервісі та покидати чергу, якщо у користувача змінилися плани.

Для перевірки якості роботи розробленого сервісу його функціонально протестовано за допомогою димного та регресійного тестування, яке пройдено на 100%. Тестування показало, що сервіс невразливий до таких розповсюджених типів атак, як SQL ін'єкції, XSS атаки та ін.

Тим самим показано, що поставлена задача виконана за допомогою розробленого сервісу.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційнотехнічними та технологічними комплексами / О.Ю. Шевченко, А.П. Ладанюк, І.В. Ельперін, С.М. Чумаченко, Л.О. Власенко, М.П. Костіков, - ISBN 978-966-612-244-8 изд. - Київ: НУХТ, 2020. - С. 234-235.

2. Наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2020, Київ, 2020 рік.

3. Wikipedia.org.com. 2020. REST — Википедия. [online] Available at: <https://ru.wikipedia.org/wiki/REST/>.

4. Habr.com. 2008. Архитектура REST. [online] Available at: <https://habr.com/ru/post/38730/>.

5. Bizzapps.ru. 2020. REST API — обзор инструментов реализации и документирования для разработчиков. [online] Available at: <https://bizzapps.ru/b/rest-api/>.

6. Kickstartpros.com. 2014. REST API - Future of SOAP. [online] Available at: <http://www.kickstartpros.com/2014/07/rest-api-future-of-soa.html>.

7. Wikibooks.org. 2020. Flask — Викиучебник. [online] Available at: <https://ru.wikibooks.org/wiki/Flask>.

8. Datbaze.ru. 2015. Что такое XSS атака и защититься от неё?. [online] Available at: <https://datbaze.ru/article/xss-ataka.html>.

9. Wikipedia.org. 2020. PostgreSQL — Вікіпедія. [online] Available at: <https://uk.wikipedia.org/wiki/PostgreSQL>.

10. Wikipedia.org. 2020. MongoDB — Вікіпедія. [online] Available at: <https://uk.wikipedia.org/wiki/MongoDB>.

11. Wikipedia.org. 2020. Apache Cassandra — Вікіпедія. [online] Available at: <https://uk.wikipedia.org/wiki/Apache_Cassandra>.

12. Wikipedia.org. 2020. Oracle Database — Вікіпедія. [online] Available at: <https://uk.wikipedia.org/wiki/Oracle_Database>.

13. Wikipedia.org. 2020. Oracle Database — Викиучебник. [online] Available at: <https://ru.wikipedia.org/wiki/Oracle_Database>.

14. Wikipedia.org. 2020. Oracle Database — Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Oracle_Database>.

15. Youtube.com. 2018. SQL vs NoSQL or MySQL vs MongoDB - YouTube. [online] Available at: <https://youtu.be/ZS_kXvOeQ5Y>.

16. Wikipedia.org. 2020. Database normalization - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Database_normalization>.

17. Роль систем масового обслуговування у підвищенні якості обслуговування клієнтів супермаркетів / Геселева Н.В., Писанець К.К., Євдоченко В.А. − К.: КНУТД, 2016. − 7 с.

18. Імовірнісні процеси: конспект лекцій / Н.І. Полтораченко. − К.: КНУБА, 2009. − 48 с.

19. Лекции теории массового обслуживания для студентов экономических специальностей очной, заочной и дистанционной форм обучения / Саакян Г.Р. − Ш.: ЮРГУЭС, 2006. − 27 с.

20. Дослідження операцій. Системи масового обслуговування. Методичні вказівки та завдання для самостійної роботи. / Суми, 2008 рік 37 ст., табл. 3, бібл. 3.

Додаток А

ЛІСТИНГИ ПРОГРАМ

Лістинг файлу main.py — головний файл де описаний REST зв'язок сайту

```
from flask import Flask, render_template, request, make_response, session, redirect, url_for, flash

from DAO import *
from wtf.form.users import *
from wtf.form.places import *
from wtf.form.statuses import *
from wtf.form.events import *
from wtf.form.createdEvents import *
from wtf.form.queues import *
import datetime

app = Flask(__name__)
app.secret_key = 'development key'


@app.route('/')
def hello():
    return render_template('index.html')


@app.route('/auth')
def auth():
    return render_template('auth.html')


@app.route('/auth/logoff')
def logoff():
    # response = make_response("Unlogged in and session with cookie was deleted")
    response = make_response(redirect('/'))
    session.pop('login', None)
    response.set_cookie("loginCookie", '', expires=0)
    return response


@app.route('/auth/login', methods=["GET", "POST"])
def login():
    form = UserLoginForm()

    if request.method == "GET":
        # if not session.has_key('login'):
        if 'login' not in session:
            login = request.cookies.get("loginCookie")
            if login is None:
                # if login is valid .. select ...
                return render_template('authLogin.html', myform=form)
            else:
```

```
            session['login'] = login
            return render_template('loggedInByCookie.html')
            # return "U R logged in by cookie"
        else:
            return render_template('loggedInBySession.html')
    if request.method == "POST":
        # form = request.form
        if not form.validate():
            return render_template('authLogin.html', myform=form)
        else:
            user = User()
            user.__enter__()
            var = user.log_in(request.form['login'], request.form['password'])

            if var == '200 OK':
                session['login'] = request.form['login']

                response = make_response(render_template('loggedIn.html'))
                # response = make_response("logged in")
                expire_date = datetime.datetime.now()
                expire_date = expire_date + datetime.timedelta(days=90)
                response.set_cookie("loginCookie", value=request.form["login"], expires=expire_date)

                return response
            else:
                flash("Wrong credentials!")
                return redirect(url_for('login'))


@app.route('/auth/registration', methods=["GET", "POST"])
def registration():
    form = UserRegistrationForm()

    if request.method == "GET":
        # if not session.has_key('login'):
        if 'login' not in session:
            login = request.cookies.get("loginCookie")
            if login is None:
                # if login is valid .. select ...
                return render_template('authRegistration.html', regForm=form)
            else:
                session['login'] = login
                return render_template('loggedInByCookie.html')
                # return "U R logged in by cookie"
        else:
            return render_template('loggedInBySession.html')
            # return "U R logged in by session"
    if request.method == "POST":
        # form = request.form
        if not form.validate():
            return render_template('authRegistration.html', regForm=form)
        else:
            user = User()
            user.__enter__()
            var = user.register(request.form['login'], request.form['password'], request.form['email'])

            # 1 create response
            if var == "200 OK":
```

```
            session['login'] = request.form['login']

            response = make_response(render_template('registered.html'))
            # response = make_response("U R registered")
            expire_date = datetime.datetime.now()
            expire_date = expire_date + datetime.timedelta(days=90)
            response.set_cookie("loginCookie", value=request.form["login"], expires=expire_date)

            return response
        else:
            response = make_response(render_template('registered.html', var=var))
            # response = make_response(var)
            return response


@app.route('/users')
def users():
    return render_template('users.html')


@app.route('/usersRead', methods=["GET", "POST"])
def usersRead():
    form = UserReadForm()

    if request.method == "GET":
        return render_template('usersRead.html', userForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('usersRead.html', userForm=form)
        else:
            user = User()
            user.__enter__()
            if request.form['select_login'] == "":
                var = user.get_users()
            else:
                var = user.get_user(request.form['select_login'])
            return render_template('usersRead.html', selectedUsers=var, userForm=form,
                                   selectedUsers_info=zip(var, range(0, len(var))))


@app.route('/usersUpdate', methods=["GET", "POST"])
def usersUpdate():
    form = UserUpdateForm()

    if request.method == "GET":
        return render_template('usersUpdate.html', userForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('usersUpdate.html', userForm=form)
        else:
            user = User()
            user.__enter__()
            var = user.update_user(request.form['old_login'], request.form['new_login'],
                                   request.form['new_password'], request.form['new_email'])
            return render_template('usersUpdate.html', updationStatus=var, userForm=form)


@app.route('/usersDelete', methods=["GET", "POST"])
```

```
def usersDelete():
    form = UserDeleteForm()

    if request.method == "GET":
        return render_template('usersDelete.html', userForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('usersDelete.html', userForm=form)
        else:
            user = User()
            user.__enter__()
            var = user.delete_user(request.form['delete_user_login'])
            return render_template('usersDelete.html', deletionStatus=var, userForm=form)


@app.route('/places')
def places():
    return render_template('places.html')


@app.route('/placesCreate', methods=["GET", "POST"])
def placesCreate():
    form = PlaceCreateForm()

    if request.method == "GET":
        return render_template('placesCreate.html', placeForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('placesCreate.html', placeForm=form)
        else:
            place = Place()
            place.__enter__()
            var = place.create_place(request.form['new_place_id'], request.form['new_address'],
                                     request.form['new_room_number'], request.form['new_schedule'])
            return render_template('placesCreate.html', creationStatus=var, placeForm=form)


@app.route('/placesRead', methods=["GET", "POST"])
def placesRead():
    form = PlaceReadForm()

    if request.method == "GET":
        return render_template('placesRead.html', placeForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('placesRead.html', placeForm=form)
        else:
            place = Place()
            place.__enter__()
            if request.form['select_place_id'] == "":
                # if request.form['combobox_id'] == "":
                var = place.get_places()
            else:
                var = place.get_place(request.form['select_place_id'])
                # var = place.get_place(request.form['combobox_id'])
            return render_template('placesRead.html', selectedPlaces=var, placeForm=form,
                                   selectedPlaces_info=zip(var, range(0, len(var))))
```

```
@app.route('/placesUpdate', methods=["GET", "POST"])
def placesUpdate():
    form = PlaceUpdateForm()

    if request.method == "GET":
        return render_template('placesUpdate.html', placeForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('placesUpdate.html', placeForm=form)
        else:
            place = Place()
            place.__enter__()
            var = place.update_place(request.form['old_place_id'], request.form['new_place_id'],
                                     request.form['new_address'], request.form['new_room_number'],
                                     request.form['new_schedule'])
            return render_template('placesUpdate.html', updationStatus=var, placeForm=form)


@app.route('/placesDelete', methods=["GET", "POST"])
def placesDelete():
    form = PlaceDeleteForm()

    if request.method == "GET":
        return render_template('placesDelete.html', placeForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('placesDelete.html', placeForm=form)
        else:
            place = Place()
            place.__enter__()
            var = place.delete_place(request.form['delete_place_id'])
            return render_template('placesDelete.html', deletionStatus=var, placeForm=form)


@app.route('/statuses')
def statuses():
    return render_template('statuses.html')


@app.route('/statusesCreate', methods=["GET", "POST"])
def statusesCreate():
    form = StatusCreateForm()

    if request.method == "GET":
        return render_template('statusesCreate.html', statusForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('statusesCreate.html', statusForm=form)
        else:
            status = Status()
            status.__enter__()
            var = status.create_status(request.form['new_status'])
            return render_template('statusesCreate.html', creationStatus=var, statusForm=form)


@app.route('/statusesRead', methods=["GET", "POST"])
def statusesRead():
```

```
    form = StatusReadForm()

    if request.method == "GET":
        return render_template('statusesRead.html', statusForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('statusesRead.html', statusForm=form)
        else:
            status = Status()
            status.__enter__()
            if request.form['select_status'] == "":
                var = status.get_statuses()
            else:
                var = status.get_status(request.form['select_status'])
            return render_template('statusesRead.html', selectedStatuses=var, statusForm=form,
                                   selectedStatuses_info=zip(var, range(0, len(var))))


@app.route('/statusesUpdate', methods=["GET", "POST"])
def statusesUpdate():
    form = StatusUpdateForm()

    if request.method == "GET":
        return render_template('statusesUpdate.html', statusForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('statusesUpdate.html', statusForm=form)
        else:
            status = Status()
            status.__enter__()
            var = status.update_status(request.form['old_status'], request.form['new_status'])
            return render_template('statusesUpdate.html', updationStatus=var, statusForm=form)


@app.route('/statusesDelete', methods=["GET", "POST"])
def statusesDelete():
    form = StatusDeleteForm()

    if request.method == "GET":
        return render_template('statusesDelete.html', statusForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('statusesDelete.html', statusForm=form)
        else:
            status = Status()
            status.__enter__()
            var = status.delete_status(request.form['delete_status'])
            return render_template('statusesDelete.html', deletionStatus=var, statusForm=form)


@app.route('/events')
def events():
    return render_template('events.html')


@app.route('/eventsCreate', methods=["GET", "POST"])
def eventsCreate():
    form = EventCreateForm()
```

```
    if request.method == "GET":
        return render_template('eventsCreate.html', eventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('eventsCreate.html', eventForm=form)
        else:
            event = Event()
            event.__enter__()
            var = event.create_event(request.form['new_event_name'])
            return render_template('eventsCreate.html', creationStatus=var, eventForm=form)


@app.route('/eventsRead', methods=["GET", "POST"])
def eventsRead():
    form = EventReadForm()

    if request.method == "GET":
        return render_template('eventsRead.html', eventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('eventsRead.html', eventForm=form)
        else:
            event = Event()
            event.__enter__()
            if request.form['select_event'] == "":
                var = event.get_events()
            else:
                var = event.get_event(request.form['select_event'])
            return render_template('eventsRead.html', selectedEvents=var, eventForm=form,
                                   selectedEvents_info=zip(var, range(0, len(var))))


@app.route('/eventsUpdate', methods=["GET", "POST"])
def eventsUpdate():
    form = EventUpdateForm()

    if request.method == "GET":
        return render_template('eventsUpdate.html', eventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('eventsUpdate.html', eventForm=form)
        else:
            event = Event()
            event.__enter__()
            var = event.update_event(request.form['old_event_name'], request.form['new_event_name'])
            return render_template('eventsUpdate.html', updationStatus=var, eventForm=form)


@app.route('/eventsDelete', methods=["GET", "POST"])
def eventsDelete():
    form = EventDeleteForm()

    if request.method == "GET":
        return render_template('eventsDelete.html', eventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('eventsDelete.html', eventForm=form)
```

```
        else:
            event = Event()
            event.__enter__()
            var = event.delete_event(request.form['delete_event'])
            return render_template('eventsDelete.html', deletionStatus=var, eventForm=form)


@app.route('/createdEvents')
def createdEvents():
    return render_template('createdEvents.html')


@app.route('/createdEventsCreate', methods=["GET", "POST"])
def createdEventsCreate():
    form = CreatedEventCreateForm()

    if request.method == "GET":
        return render_template('createdEventsCreate.html', createdEventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('createdEventsCreate.html', createdEventForm=form)
        else:
            createdEvent = CreatedEvent()
            createdEvent.__enter__()
            var = createdEvent.create_created_event(request.form['new_place_id'], request.form['new_event_name'],
                                                    request.form['new_date_creation_event'])
            return render_template('createdEventsCreate.html', creationStatus=var, createdEventForm=form)


@app.route('/createdEventsRead', methods=["GET", "POST"])
def createdEventsRead():
    form = CreatedEventReadForm()

    if request.method == "GET":
        return render_template('createdEventsRead.html', createdEventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('createdEventsRead.html', createdEventForm=form)
        else:
            createdEvent = CreatedEvent()
            createdEvent.__enter__()
            if request.form['select_place_id'] == "":
                var = createdEvent.get_created_events_0()
            else:
                if request.form['select_event_name'] == "":
                    var = createdEvent.get_created_events_1(request.form['select_place_id'])
                else:
                    if request.form['select_date_creation_event'] == "":
                        var = createdEvent.get_created_events_2(request.form['select_place_id'],
                                                                request.form['select_event_name'])
                    else:
                        var = createdEvent.get_created_events_3(request.form['select_place_id'],
                                                                request.form['select_event_name'],
                                                                request.form['select_date_creation_event'])

            return render_template('createdEventsRead.html', selectedCreatedEvents=var, createdEventForm=form,
                                   selectedCreatedEvents_info=zip(var, range(0, len(var))))
```

```
@app.route('/createdEventsUpdate', methods=["GET", "POST"])
def createdEventsUpdate():
    form = CreatedEventUpdateForm()

    if request.method == "GET":
        return render_template('createdEventsUpdate.html', createdEventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('createdEventsUpdate.html', createdEventForm=form)
        else:
            createdEvent = CreatedEvent()
            createdEvent.__enter__()
            var = createdEvent.update_created_event(request.form['old_place_id'], request.form['old_event_name'],
                                        request.form['old_date_creation_event'],
                                        request.form['new_place_id'],
                                        request.form['new_event_name'],
                                        request.form['new_date_creation_event'])
            return render_template('createdEventsUpdate.html', updationStatus=var, createdEventForm=form)


@app.route('/createdEventsDelete', methods=["GET", "POST"])
def createdEventsDelete():
    form = CreatedEventDeleteForm()

    if request.method == "GET":
        return render_template('createdEventsDelete.html', createdEventForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('createdEventsDelete.html', createdEventForm=form)
        else:
            createdEvent = CreatedEvent()
            createdEvent.__enter__()
            var = createdEvent.delete_created_event(request.form['delete_place_id'], request.form['delete_event_name'],
                                        request.form['delete_date_creation_event'])
            return render_template('createdEventsDelete.html', deletionStatus=var, createdEventForm=form)


@app.route('/queues')
def queues():
    return render_template('queues.html')


@app.route('/queuesCreate', methods=["GET", "POST"])
def queuesCreate():
    form = QueueCreateForm()

    if request.method == "GET":
        return render_template('queuesCreate.html', queueForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('queuesCreate.html', queueForm=form)
        else:
            queue = Queue()
            queue.__enter__()
            var = queue.create_queue(request.form['new_status'], request.form['new_user_login'],
                                request.form['new_place_id'], request.form['new_event_name'],
                                request.form['new_date_creation_event'],
```

```
                                request.form['new_date_request_creation'])
            return render_template('queuesCreate.html', creationStatus=var, queueForm=form)


@app.route('/queuesRead', methods=["GET", "POST"])
def queuesRead():
    form = QueueReadForm()

    if request.method == "GET":
        return render_template('queuesRead.html', queueForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('queuesRead.html', queueForm=form)
        else:
            queue = Queue()
            queue.__enter__()
            if request.form['select_place_id'] == "":
                var = queue.get_queues_0()
            else:
                if request.form['select_user_login'] == "":
                    var = queue.get_queues_1(request.form['select_place_id'])
                else:
                    if request.form['select_status'] == "":
                        var = queue.get_queues_2(request.form['select_place_id'],
                                                 request.form['select_user_login'])
                    else:
                        if request.form['select_event_name'] == "":
                            var = queue.get_queues_3(request.form['select_place_id'],
                                                     request.form['select_user_login'],
                                                     request.form['select_status'])
                        else:
                            if request.form['select_date_creation_event'] == "":
                                var = queue.get_queues_4(request.form['select_place_id'],
                                                         request.form['select_user_login'],
                                                         request.form['select_status'],
                                                         request.form['select_event_name'])
                            else:
                                if request.form['select_date_request_creation'] == "":
                                    var = queue.get_queues_5(request.form['select_place_id'],
                                                             request.form['select_user_login'],
                                                             request.form['select_status'],
                                                             request.form['select_event_name'],
                                                             request.form['select_date_creation_event'])
                                else:
                                    var = queue.get_queues_6(request.form['select_place_id'],
                                                             request.form['select_user_login'],
                                                             request.form['select_status'],
                                                             request.form['select_event_name'],
                                                             request.form['select_date_creation_event'],
                                                             request.form['select_date_request_creation'])
            return render_template('queuesRead.html', selectedQueues=var, queueForm=form,
                                   selectedQueues_info=zip(var, range(0, len(var))))


@app.route('/queuesUpdate', methods=["GET", "POST"])
def queuesUpdate():
    form = QueueUpdateForm()
```

```
    if request.method == "GET":
        return render_template('queuesUpdate.html', queueForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('queuesUpdate.html', queueForm=form)
        else:
            queue = Queue()
            queue.__enter__()
            var = queue.update_queue(request.form['old_status'], request.form['old_user_login'],
                                     request.form['old_place_id'], request.form['old_event_name'],
                                     request.form['old_date_creation_event'],
                                     request.form['old_date_request_creation'],
                                     request.form['new_status'], request.form['new_user_login'],
                                     request.form['new_place_id'], request.form['new_event_name'],
                                     request.form['new_date_creation_event'],
                                     request.form['new_date_request_creation'])
            return render_template('queuesUpdate.html', updationStatus=var, queueForm=form)


@app.route('/queuesDelete', methods=["GET", "POST"])
def queuesDelete():
    form = QueueDeleteForm()

    if request.method == "GET":
        return render_template('queuesDelete.html', queueForm=form)
    if request.method == "POST":
        if not form.validate():
            return render_template('queuesDelete.html', queueForm=form)
        else:
            queue = Queue()
            queue.__enter__()
            var = queue.delete_queue(request.form['delete_status'], request.form['delete_user_login'],
                                     request.form['delete_place_id'], request.form['delete_event_name'],
                                     request.form['delete_date_creation_event'],
                                     request.form['delete_date_request_creation'])
            return render_template('queuesDelete.html', deletionStatus=var, queueForm=form)


app.run(debug=True)
```

Лістинг файлу DAO.py — файл де описана робота фреймворку з пакетами бази даних

```
import cx_Oracle as cx_Oracle


class User:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()
```

```
    def register(self, login, password, email):
        result = self.__cursor.callfunc("USER_PACKAGE.REGISTER", cx_Oracle.STRING, [login, password, email])
        return result

    def log_in(self, login, password):
        result = self.__cursor.callfunc("USER_PACKAGE.LOG_IN", cx_Oracle.STRING, [login, password])
        return result

    def get_user(self, select_login):
        query = 'select * from table ( USER_PACKAGE.GET_USERS(:LOGIN) )'
        var = self.__cursor.execute(query, LOGIN=select_login)
        return var.fetchall()

    def get_users(self):
        query = 'select * from table (USER_PACKAGE.GET_USERS())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def update_user(self, oldLogin, newLogin, newPassword, newEmail):
        result = self.__cursor.callfunc("USER_PACKAGE.UPDATE_USER", cx_Oracle.STRING,
                                        [oldLogin, newLogin, newPassword, newEmail])
        return result

    def delete_user(self, login):
        result = self.__cursor.callfunc("USER_PACKAGE.DELETE_USER", cx_Oracle.STRING, [login])
        return result


class Place:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()

    def create_place(self, newPlaceID, newAddress, newRoomNumber, newSchedule):
        result = self.__cursor.callfunc("PLACE_PACKAGE.CREATE_PLACE", cx_Oracle.STRING,
                                        [newPlaceID, newAddress, newRoomNumber, newSchedule])
        return result

    def get_place(self, p_id):
        query = 'select * from table ( PLACE_PACKAGE.GET_PLACES(:PLACE_ID) )'
        var = self.__cursor.execute(query, PLACE_ID=p_id)
        return var.fetchall()

    def get_places(self):
        query = 'select * from table (PLACE_PACKAGE.GET_PLACES())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def get_places_id(self):
        query = 'select * from table (PLACE_PACKAGE.GET_PLACES_ID())'
        var = self.__cursor.execute(query)
        return var.fetchall()
```

```
    def update_place(self, oldPlaceID, newPlaceID, newAddress, newRoomNumber, newSchedule):
        result = self.__cursor.callfunc("PLACE_PACKAGE.UPDATE_PLACE", cx_Oracle.STRING,
                                        [oldPlaceID, newPlaceID, newAddress, newRoomNumber, newSchedule])
        return result

    def delete_place(self, placeID):
        result = self.__cursor.callfunc("PLACE_PACKAGE.DELETE_PLACE", cx_Oracle.STRING, [placeID])
        return result


class Event:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()

    def create_event(self, NEW_E_NAME):
        result = self.__cursor.callfunc("EVENT_PACKAGE.CREATE_EVENT", cx_Oracle.STRING, [NEW_E_NAME])
        return result

    def get_event(self, E_NAME):
        query = 'select * from table ( EVENT_PACKAGE.GET_EVENTS(:EVENT_NAME) )'
        var = self.__cursor.execute(query, EVENT_NAME=E_NAME)
        return var.fetchall()

    def get_events(self):
        query = 'select * from table (EVENT_PACKAGE.GET_EVENTS())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def update_event(self, OLD_E_NAME, NEW_E_NAME):
        result = self.__cursor.callfunc("EVENT_PACKAGE.UPDATE_EVENT", cx_Oracle.STRING, [OLD_E_NAME,
NEW_E_NAME])
        return result

    def delete_event(self, E_NAME):
        result = self.__cursor.callfunc("EVENT_PACKAGE.DELETE_EVENT", cx_Oracle.STRING, [E_NAME])
        return result


class Status:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()

    def create_status(self, NEW_STATUS):
        result = self.__cursor.callfunc("STATUS_PACKAGE.CREATE_STATUS", cx_Oracle.STRING,
[NEW_STATUS])
        return result
```

```
    def get_status(self, S_STATUS):
        query = 'select * from table ( STATUS_PACKAGE.GET_STATUSES(:STATUS) )'
        var = self.__cursor.execute(query, STATUS=S_STATUS)
        return var.fetchall()

    def get_statuses(self):
        query = 'select * from table (STATUS_PACKAGE.GET_STATUSES())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def update_status(self, OLD_STATUS, NEW_STATUS):
        result = self.__cursor.callfunc("STATUS_PACKAGE.UPDATE_STATUS", cx_Oracle.STRING,
[OLD_STATUS, NEW_STATUS])
        return result

    def delete_status(self, S_STATUS):
        result = self.__cursor.callfunc("STATUS_PACKAGE.DELETE_STATUS", cx_Oracle.STRING, [S_STATUS])
        return result


class CreatedEvent:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()

    def create_created_event(self, NEW_PLACE_ID, NEW_EVENT_NAME, NEW_DATE_CREATION_EVENT):
        result = self.__cursor.callfunc("CREATED_EVENT_PACKAGE.CREATE_CREATED_EVENT",
cx_Oracle.STRING,
                                        [NEW_PLACE_ID, NEW_EVENT_NAME, NEW_DATE_CREATION_EVENT])
        return result

    def get_created_events_0(self):
        query = 'select * from table (CREATED_EVENT_PACKAGE.GET_CREATED_EVENTS())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def get_created_events_1(self, select_status):
        query = 'select * from table ( CREATED_EVENT_PACKAGE.GET_CREATED_EVENTS(:STATUS) )'
        var = self.__cursor.execute(query, STATUS=select_status)
        return var.fetchall()

    def get_created_events_2(self, select_status, select_user_login):
        query = 'select * from table ( CREATED_EVENT_PACKAGE.GET_CREATED_EVENTS(:STATUS,
:USER_LOGIN) )'
        var = self.__cursor.execute(query, STATUS=select_status, USER_LOGIN=select_user_login)
        return var.fetchall()

    def get_created_events_3(self, select_place_id, select_event_name, select_date_creation_event):
        query = 'select * from table ( CREATED_EVENT_PACKAGE.GET_CREATED_EVENTS(:PLACE_ID,
:EVENT_NAME, :DATE_CREATION_EVENT) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, EVENT_NAME=select_event_name,
                                    DATE_CREATION_EVENT=select_date_creation_event)
```

```
        return var.fetchall()

    def update_created_event(self, OLD_PLACE_ID, OLD_EVENT_NAME, OLD_DATE_CREATION_EVENT,
NEW_PLACE_ID, NEW_EVENT_NAME,
                             NEW_DATE_CREATION_EVENT):
        result = self.__cursor.callfunc("CREATED_EVENT_PACKAGE.UPDATE_CREATED_EVENT",
cx_Oracle.STRING,
                                        [OLD_PLACE_ID, OLD_EVENT_NAME, OLD_DATE_CREATION_EVENT,
NEW_PLACE_ID,
                                         NEW_EVENT_NAME, NEW_DATE_CREATION_EVENT])
        return result

    def delete_created_event(self, CE_PLACE_ID, CE_EVENT_NAME, CE_DATE_CREATION_EVENT):
        result = self.__cursor.callfunc("CREATED_EVENT_PACKAGE.DELETE_CREATED_EVENT",
cx_Oracle.STRING,
                                        [CE_PLACE_ID, CE_EVENT_NAME, CE_DATE_CREATION_EVENT])
        return result


class Queue:
    def __enter__(self):
        self.__db = cx_Oracle.connect('ledoff', 'Password1', "localhost:1521/orcl")
        self.__cursor = self.__db.cursor()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.__cursor.close()
        self.__db.close()

    def create_queue(self, NEW_STATUS, NEW_USER_LOGIN, NEW_PLACE_ID, NEW_EVENT_NAME,
NEW_DATE_CREATION_EVENT,
                     NEW_DATE_REQUEST_CREATION):
        result = self.__cursor.callfunc("QUEUE_PACKAGE.CREATE_QUEUE", cx_Oracle.STRING,
                                        [NEW_STATUS, NEW_USER_LOGIN, NEW_PLACE_ID, NEW_EVENT_NAME,
                                         NEW_DATE_CREATION_EVENT, NEW_DATE_REQUEST_CREATION])
        return result

    def get_queues_0(self):
        query = 'select * from table (QUEUE_PACKAGE.GET_QUEUES())'
        var = self.__cursor.execute(query)
        return var.fetchall()

    def get_queues_1(self, select_place_id):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id)
        return var.fetchall()

    def get_queues_2(self, select_place_id, select_user_login):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID, :USER_LOGIN) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, USER_LOGIN=select_user_login)
        return var.fetchall()

    def get_queues_3(self, select_place_id, select_user_login, select_status):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID, :USER_LOGIN, :STATUS) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, USER_LOGIN=select_user_login,
STATUS=select_status)
        return var.fetchall()
```

```
    def get_queues_4(self, select_place_id, select_user_login, select_status, select_event_name):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID, :USER_LOGIN, :STATUS,
:EVENT_NAME) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, USER_LOGIN=select_user_login,
STATUS=select_status,
                                    EVENT_NAME=select_event_name)
        return var.fetchall()

    def get_queues_5(self, select_place_id, select_user_login, select_status, select_event_name,
                     select_date_creation_event):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID, :USER_LOGIN, :STATUS,
:EVENT_NAME, :DATE_CREATION_EVENT) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, USER_LOGIN=select_user_login,
STATUS=select_status,
                                    EVENT_NAME=select_event_name, DATE_CREATION_EVENT=select_date_creation_event)
        return var.fetchall()

    def get_queues_6(self, select_place_id, select_user_login, select_status, select_event_name,
                     select_date_creation_event, select_date_request_creation):
        query = 'select * from table ( QUEUE_PACKAGE.GET_QUEUES(:PLACE_ID, :USER_LOGIN, :STATUS,
:EVENT_NAME, :DATE_CREATION_EVENT, :DATE_REQUEST_CREATION) )'
        var = self.__cursor.execute(query, PLACE_ID=select_place_id, USER_LOGIN=select_user_login,
STATUS=select_status,
                                    EVENT_NAME=select_event_name, DATE_CREATION_EVENT=select_date_creation_event,
                                    DATE_REQUEST_CREATION=select_date_request_creation)
        return var.fetchall()

    def update_queue(self, OLD_STATUS, OLD_USER_LOGIN, OLD_PLACE_ID, OLD_EVENT_NAME,
OLD_DATE_CREATION_EVENT,
                     OLD_DATE_REQUEST_CREATION, NEW_STATUS, NEW_USER_LOGIN, NEW_PLACE_ID,
NEW_EVENT_NAME,
                     NEW_DATE_CREATION_EVENT, NEW_DATE_REQUEST_CREATION):
        result = self.__cursor.callfunc("QUEUE_PACKAGE.UPDATE_QUEUE", cx_Oracle.STRING,
                                        [OLD_STATUS, OLD_USER_LOGIN, OLD_PLACE_ID, OLD_EVENT_NAME,
                                         OLD_DATE_CREATION_EVENT,
                                         OLD_DATE_REQUEST_CREATION, NEW_STATUS, NEW_USER_LOGIN,
NEW_PLACE_ID,
                                         NEW_EVENT_NAME, NEW_DATE_CREATION_EVENT,
NEW_DATE_REQUEST_CREATION])
        return result

    def delete_queue(self, Q_STATUS, Q_USER_LOGIN, Q_PLACE_ID, Q_EVENT_NAME,
Q_DATE_CREATION_EVENT,
                     Q_DATE_REQUEST_CREATION):
        result = self.__cursor.callfunc("QUEUE_PACKAGE.DELETE_QUEUE", cx_Oracle.STRING,
                                        [Q_STATUS, Q_USER_LOGIN, Q_PLACE_ID, Q_EVENT_NAME,
Q_DATE_CREATION_EVENT,
                                         Q_DATE_REQUEST_CREATION])
        return result
```

Лістинг файлу users.html — файл де описана розмітка сторінки для роботи із сутністю користувач

```
<html lang="en">
```

```
<head>
   <title>Users</title>
   <meta charset="utf-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
   <!--<link rel="stylesheet" href="css/floating-labels.css">-->
   <!--<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></script>-->
   <!--<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"></script>-->
   <!--<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>-->
</head>
<body>
<nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
   <a class="navbar-brand" href="/">Home page</a>
</nav>
<br><br><br><br><br>
<div class="container">
   <div class="row">
      <div class="col-sm">
         <p><a class="btn btn-success" href="/usersRead" role="button">Users read</a></p>
      </div>
      <div class="col-sm">
         <p><a class="btn btn-success" href="/usersUpdate" role="button">Users update</a></p>
      </div>
      <div class="col-sm">
         <p><a class="btn btn-success" href="/usersDelete" role="button">Users delete</a></p>
      </div>
   </div>
   <hr>
</div>

</body>
</html>
```

Лістинг файлу users.py — файл де описана форма користувача з відповідною валідацією для клієнта

```
# from flask_wtf import FlaskForm
from flask_wtf import Form
from wtforms import StringField, SubmitField, validators


class UserRegistrationForm(Form):
   login = StringField("Login", [validators.DataRequired("Required"),
                                 validators.regexp('^[^";]{4,20}$', flags=0,
                                                   message='Please enter like this [^";]{4,20}')])
   password = StringField("Password", [validators.DataRequired("Required"),
                                       validators.regexp('^[^";]{4,20}$', flags=0,
                                                         message='Please enter like this [^";]{4,20}')])
   email = StringField("Email")
   submit = SubmitField("Register user")


class UserLoginForm(Form):
```

```
    login = StringField("Login", [validators.DataRequired("Required")])
    password = StringField("Password", [validators.DataRequired("Required")])
    submit = SubmitField("Log in")


class UserReadForm(Form):
    select_login = StringField("Login")
    submit = SubmitField("Select one or all users")


class UserUpdateForm(Form):
    old_login = StringField("Old login ", [validators.DataRequired("Required"),
                                validators.regexp('^[^";]{4,20}$', flags=0,
                                            message='Please enter like this [^";]{4,20}')])
    new_login = StringField("New login ", [validators.DataRequired("Required"),
                                validators.regexp('^[^";]{4,20}$', flags=0,
                                            message='Please enter like this [^";]{4,20}')])
    new_password = StringField("New password ", [validators.DataRequired("Required"),
                                    validators.regexp('^[^";]{4,20}$', flags=0,
                                                message='Please enter like this [^";]{4,20}')])
    new_email = StringField("New email ")
    submit = SubmitField("Update user")


class UserDeleteForm(Form):
    delete_user_login = StringField("Delete user with login ", [validators.DataRequired("Required")])
    submit = SubmitField("Delete user")
```

ДОДАТОК Б
ГРАФІЧНИЙ МАТЕРІАЛ

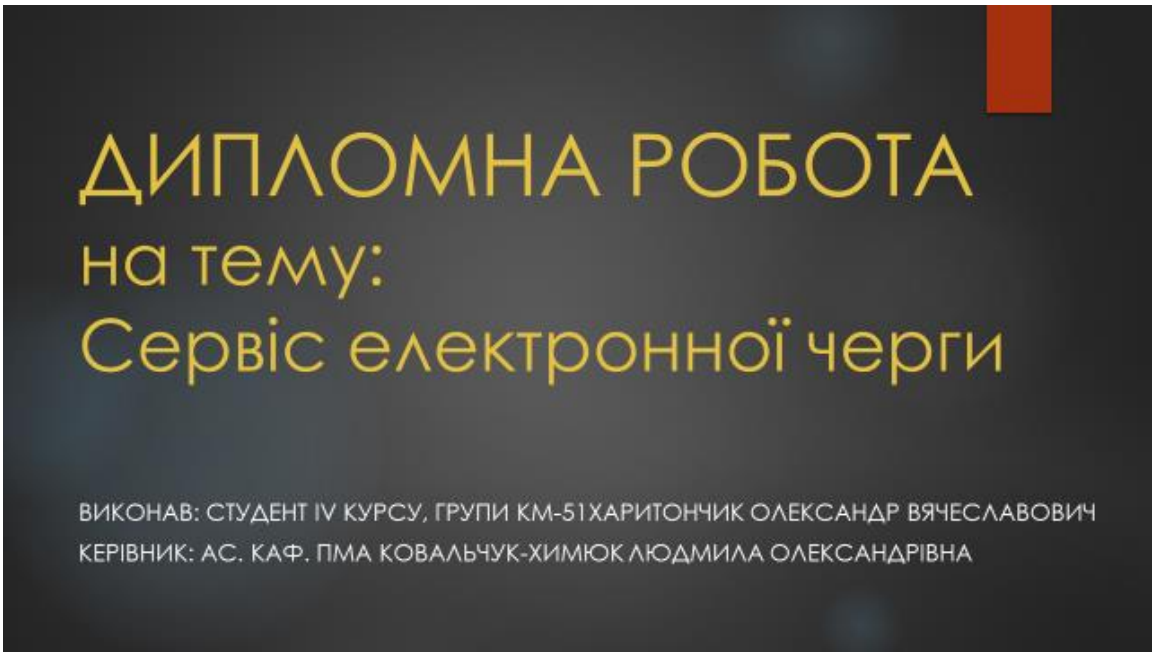

Рисунок 41 – Слайд 1

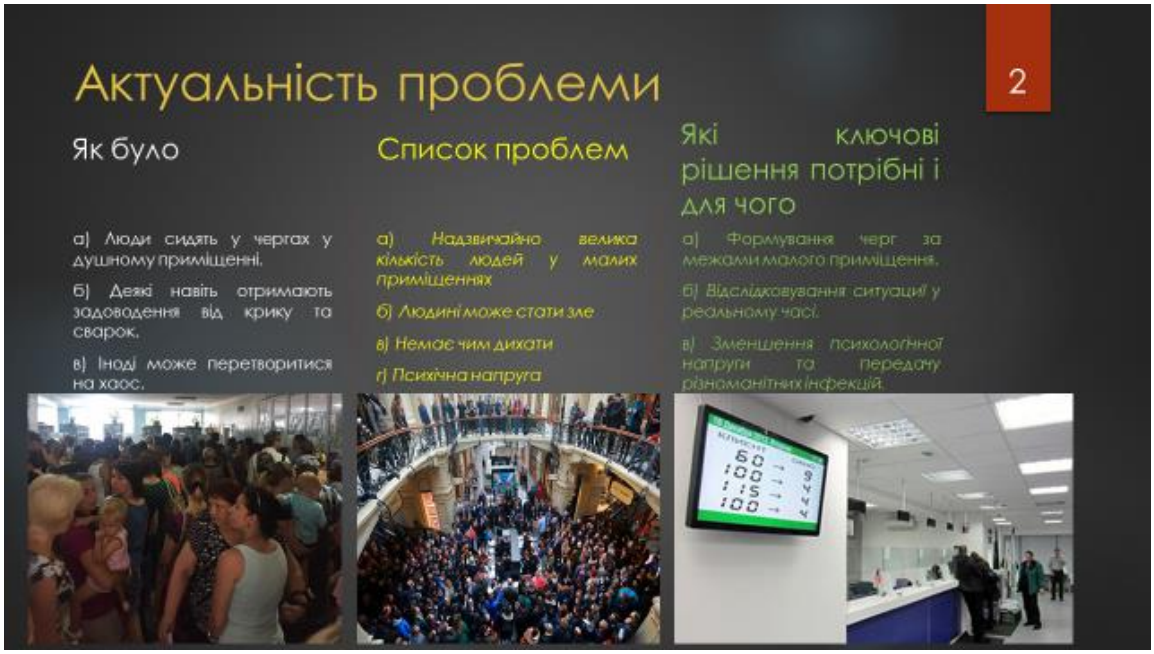

Рисунок 42 – Слайд 2

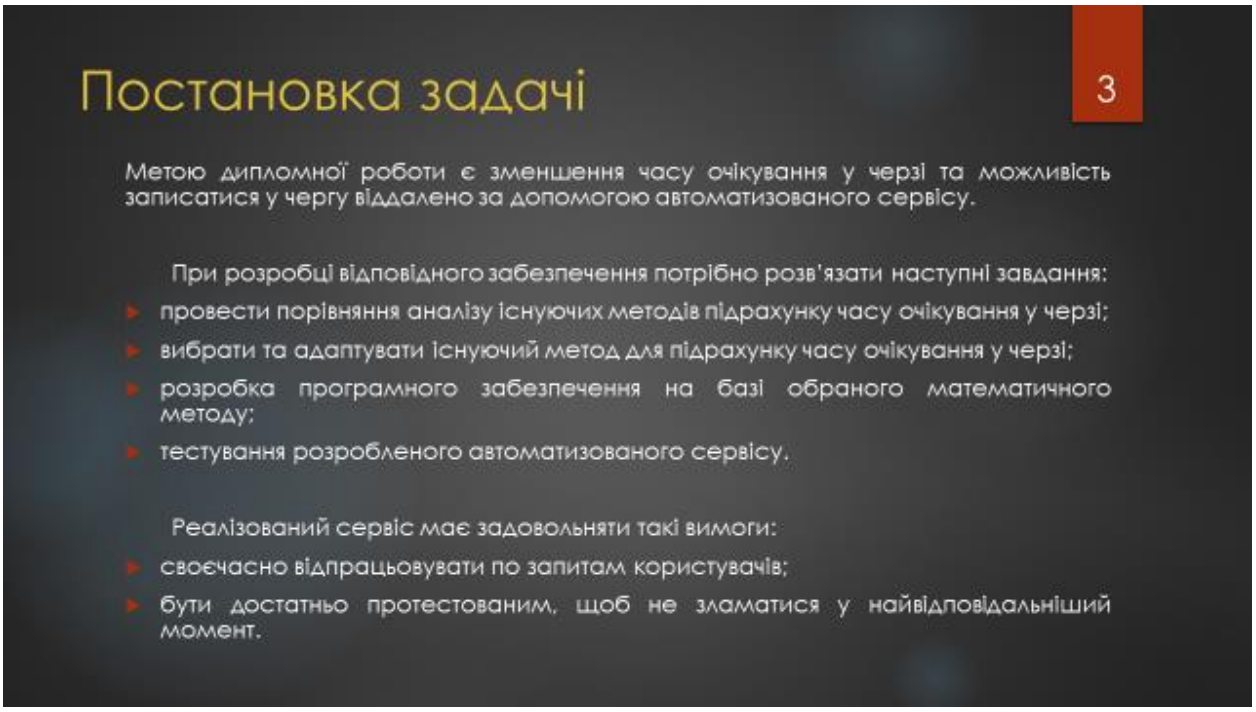

Рисунок 43 – Слайд 3

Рисунок 44 – Слайд 4

Рисунок 45 – Слайд 5

Рисунок 46 – Слайд 6

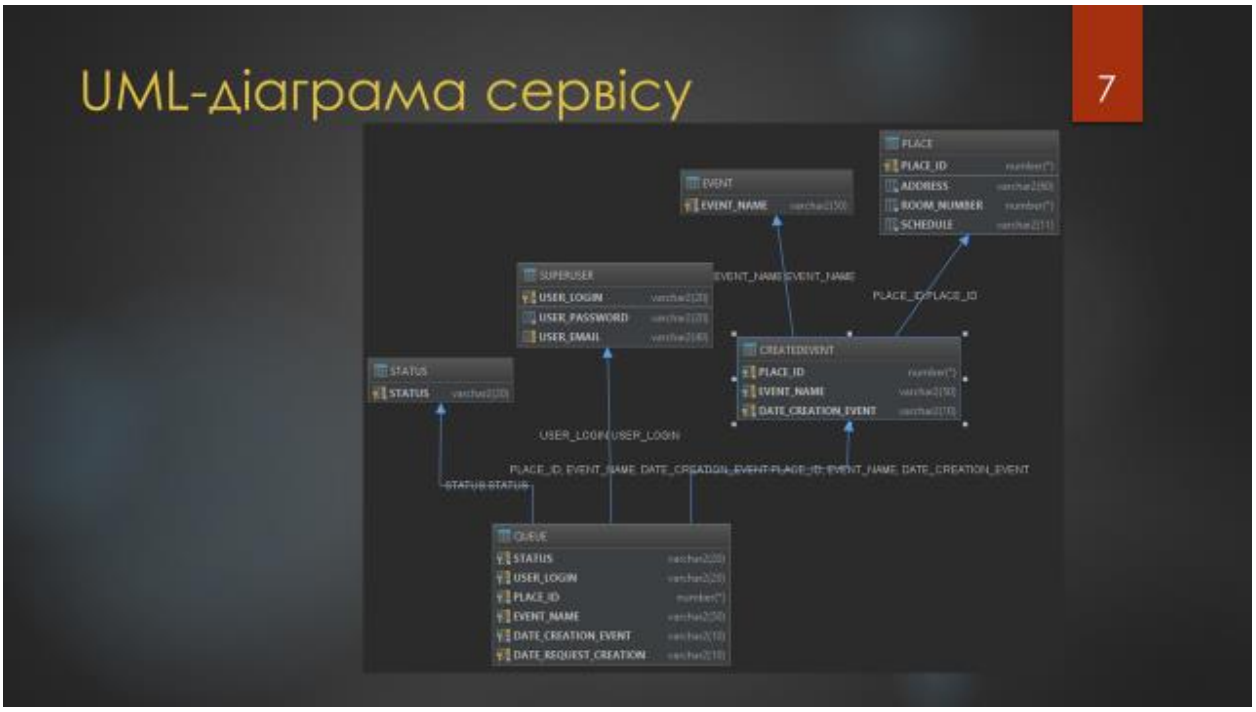

Рисунок 47 – Слайд 7

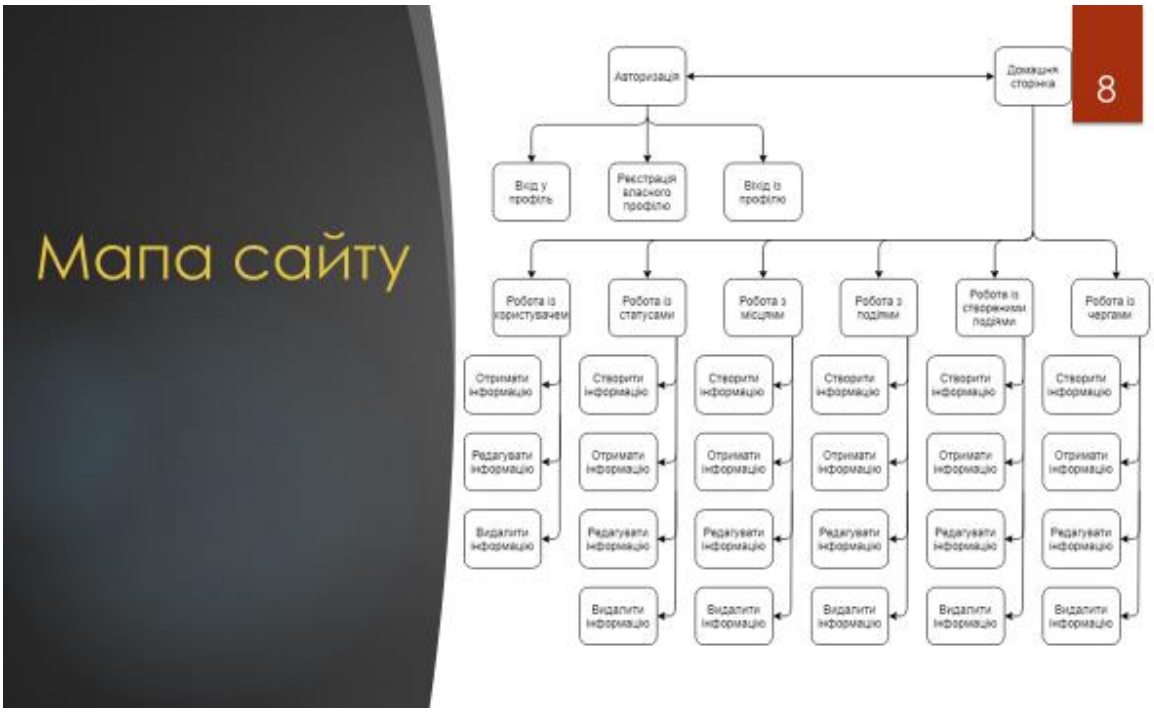

Рисунок 48 – Слайд 8

Рисунок 49 – Слайд 9

Дослідження ефективності системи 10

Нехай інтенсивність надходження заявок λ становить 10 одиниць на годину, а тривалість обслуговування лікаря $t_{об}$ = 8 хвилин.

Основні характеристики обслуговування заявок на огляд поліклінікою залежно від кількості лікарів:

Характеристика	Кількість каналів (лікарів)			
	1	2	3	4
Відносна пропускна здатність (Q)	0,42	0,72	0,89	0,96
Абсолютна пропускна здатність (A)	4,28	7,24	8,9	9,65

Отже, це означає, що за годину будуть обслуговуватися в середньому 4,28; 7.24; 8.9; 9.65 заявки (А), тобто 42, 72, 89 та 96 (%) звернувшихся за допомогою її отримають, при $n = \overline{1..4}$ відповідно.

Рисунок 50 – Слайд 10

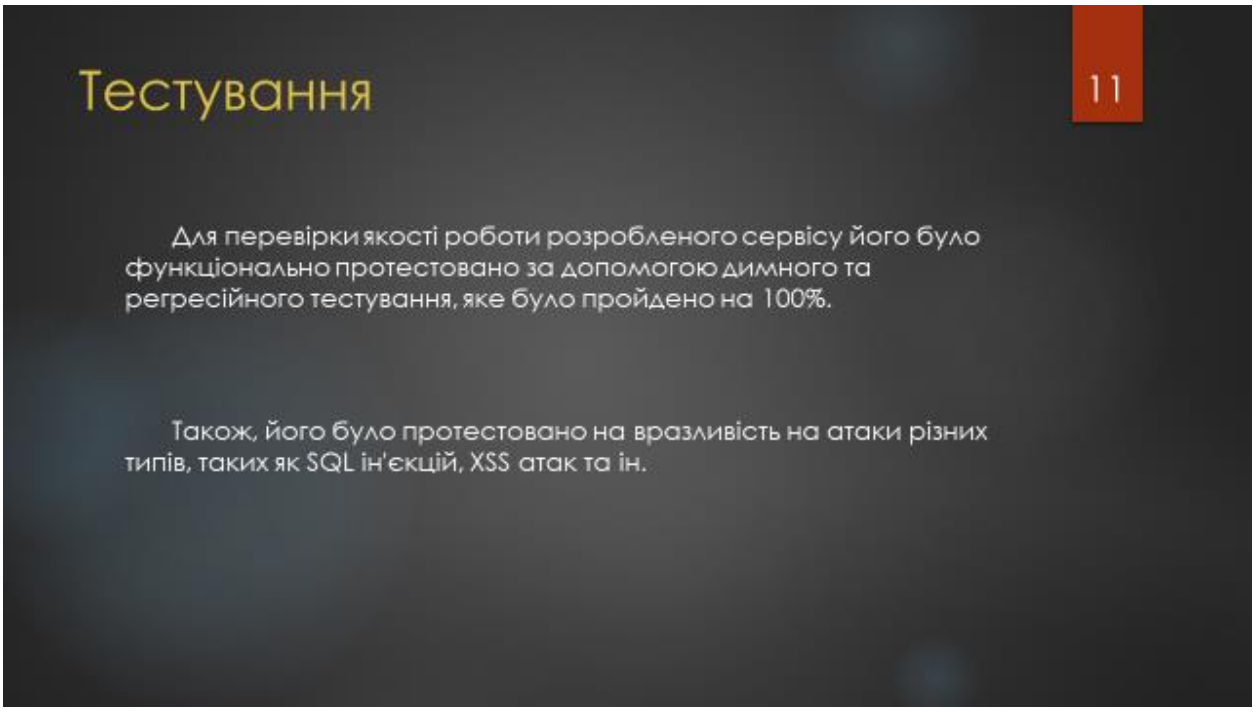

Рисунок 51 – Слайд 11

Рисунок 52 – Слайд 12

Рисунок 53 – Слайд 13